1984

Implementation of a serial communication process for a fault tolerant, real time, multitransputer operating system.

Selcuk, Zafer

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION OF A SERIAL COMMUNICATION PROCESS FOR A FAULT TOLERANT, REAL TIME, MULTITRANSPUTER OPERATING SYSTEM

by

Zafer Selcuk

December 1984

Thesis Advisor: Uno R. Kodres

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Implementation of a Serial Communication Process for a Fault Tolerant, Real Time, Multitransputer Operating System

5. TYPE OF REPORT & PERIOD COVERED

Master's thesis
December 1984

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Zafer Selcuk

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93943

12. REPORT DATE

December 1984

13. NUMBER OF PAGES

108

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Transputer, Occam, Concurrent Processing,
Multi Computer, Real Time Systems,
Fault Tolerance, Quadruple Modular Redundancy

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of this thesis is to study a more reliable and faster microcomputer systems for real time applications. The new product of VLSI technology, IMS T424 Transputer is used for the prototype design of this multicomputer system. (cont.)

ABSTRACT (Continued) # 20

The multicomputer system's design with the fault tolerance feature, is emphasised in this thesis. Also a serial communication subsystem has been implemented using the VAX 11/780 VMS system for the proposed fault tolerant real time multitransputer operating system.

Approved for public release; distribution is unlimited.

Implementation of a Serial Communication Process for a Fault Tolerant, Real Time, Multi Transputer Operating System

by

Zafer Selcuk
Lieutenant J.G., Turkish Navy
B.S., Turkish Naval Academy, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

ABSTRACT

The purpose of this thesis is to study a more reliable and faster microcomputer system for real time applications. The new product of VLSI technology, IMS T424 Transputer is used for the prototype design of this multicomputer system.

The multicomputer system's design with the fault tolerance feature, is emphasised in this thesis. Also a serial communication subsystem has been implemented using the VAX 11/780 VMS system for the proposed fault tolerant real time multitransputer operating system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to dedicate this work to my wife Sibel.

Also I would like to thank Prof. Uno R. Kodres for his guidance, and Prof. Bruce J. MacLennan and the staff of the Computer Science department for their help and understanding.

DISCLAIMER

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis will be listed below following the firm holding the trademark.

INMOS Limited Corporation, Colorado Springs, Colorado

INMOS
OCCAM
Transputer
IMS T424
Inmos links
IMS 1400 (RAM)
IMS 3630 (RAM)
IMS 2620 (EPROM)

I. INTRODUCTION

A. BACKGROUND

With the rapid advances in computer technologies both hardware and software, computer systems are becoming more complex. Together with its wide application, increasing system reliability is of vital importance. [Ref. 1] Providing fault tolerance is one of the attractive solutions.

Specific reasons for the use of fault tolerance exist in several areas of present applications. Examples are:

1. Safety related failures, such as medical support or defense systems.
2. Failure and short outage causes economic penalties, such as telephone , banking , and time sharing systems.
3. Environments where manual maintenance is not possible, such as space, sea or undersea. [Ref. 2]

In general fault tolerance consist of three sequential steps.

1. Detection
2. Diagnosis
3. Recovery

The detected error is analyzed to isolate the fault cause. Recovering from the failure requires at least a back-up system such as secondary memory, a spare processor and system's buses in real time systems. As a consequence, computing machines become larger and more complex. During normal fault free operation, fault tolerance does not provide any performance advantages, but it is the insurance of the logic machine against disruptive physical events. [Ref. 3]

B. OBJECTIVES

This thesis is intended to focus on the area of implementing a serial bus communication subsystem as a secondary bus structure of the real time multitransputer system to provide for the fault tolerance requirement.

C. MOTIVATION

To explain the motivation of this thesis, it is better to give some quantitative information about the computing capacity and the performance of the transputer, IMS T424 [Ref. 4].

TABLE 1
HARDWARE CAPABILITY OF THE IMS T424

processor	32 bits (4.9 * 10^9 integer)
processing speed . . .	10 MIPS (950 nanosec. mult.)
memory capacity	32 bit address bus 4 GBytes
built in memory	4 KBytes RAM. 80 MBytes/sec
serial bus	4 INMOS links 1.5 Mbytes/sec
parallel bus	25 Mbytes/sec max. transfer
peripheral interface .	8 bits bidir. 4 Mbytes/sec.
power dissipation . . .	0.9 Watts
physical	45 mm^2 chip mounted in an 84 contact leadless chip carrier.

At the first glance the quantities in Table 1 seem very attractive just to use the transputer as a substitute processor in an ordinary uniprocessor system.

The important hardware feature of the transputer is the four bidirectional serial communication channels. Therefore it is possible to obtain multiple communication paths between two elements of the multitransputer system, using some structures of the transputers. These paths provide the graceful degradation for redundant multitransputer systems. In other words, we can simply by-pass the failed element in the multisystem, and processing continues with other elements of the system.

Another important feature of the transputer, is that it provides excellent hardware for concurrent processing. In other words the transputer has been designed using a reduced instruction set architecture which implements the OCCAM concurrent programming language efficiently [Ref. 5].

D. THESIS STRUCTURE

The introduction just presented is designed to provide the reader with a brief look at fault tolerance and in particular to the development decisions on which a serial bus communication structure based.

Chapter II will describe the hardware architecture and the capabilities of the multitransputer systems. Chapter III will provide a brief explanation of the OCCAM concurrent processing language and its features. Chapter IV will outline the fault tolerant system structure and details of the serial bus communication process. In Chapter V the performance of the serial bus communication is evaluated and compared with fault free system performance.

The final chapter presents conclusions and observations that resulted from this thesis effort and suggestions for further research. Eight appendices are also provided that give detailed descriptions of the subprocesses of the serial bus communication program and its implementation.

II. HARDWARE FEATURES

A. WHAT IS THE TRANSPUTER ?

Transputer, or a transistor computer, is a single chip computer which provides a direct implementation for the process model of computing, in which each process is an independent computation with its own data and program. The processes are executed in a time shared mode on the transputer and special instructions are provided to support the process model of communication.

The term "Transputer" also reflects the device's ability to be used as a system building block. The word is derived from 'transistor' and 'computer' since the transputer is both a computer on a chip and a silicon component like a transistor. Just as the use of logic gates and Boolean Algebra provides the design methodology of the present electronic systems, so the transputer together with formal rules of OCCAM provides the design methodology for future concurrent systems.

The detailed descriptions of the components of the transputer: processor, memory, links and peripheral interface will be introduced in next four subsection respectively.

1. Processor

IMS T424 transputer has a 32 bit processor with an instruction execution rate of 10 MIPS. Typical instructions carried out by the processor and their execution times are given in Table 2 .

The processor is optimized to implement the OCCAM programming language. It is designed for performance and

TABLE 2

IMS T424 INSTRUCTION LIST AND EXECUTION TIMES

INSTRUCTION	EXECUTION TIME (nano sec.)
arithmetic operands	
+, -	50
multiplication	950
division	1950
remainder	1950
comparison operators	
=, ≠, >, ≥, <, ≤	100
logical operators	
AND, OR	50
shifting	
<<[n], >>[n]	50n+50
identifiers	
variable	120
vector variable	160
expression evaluation	
constant	70
parenthesis	50
constructor	
sequential	0
parallel	450n-200
alternative	600n+600
branch (IF)	150n
repetitive (WHILE)	200
primitives	
! (output) ? (input)	625
assignment	0

efficiency, which is achieved by designing the instruction set to simplify instruction decoding, by having a minimum number of special registers, by incorporating useful functions into the registers and by the use of an evaluation stack. This approach simplifies compiler design, since all the operands are in an uniformly addressed data space and it also gives a fast process switch time.

The instruction set is compact. This is achieved by separating data access from manipulation. Most instructions are one byte long and are divided into two four bit fields: function and operand. A sequence of bytes can be used to extend the instruction in units of four bits up to a word

length, enabling both functions and operands to be frequency encoded.

The processor is designed to execute high level languages efficiently and will normally be programmed using OCCAM or an industry standard languages such as C or PASCAL.

a. Concurrent Processing

The processor executes programs sequentially. It implements parallel processes by sharing its time between the set of processes which are active at any instant. A process is active when it is not waiting for input or output.

The currently executed process runs until it has to wait for communication. When this happens, the process is set inactive and the next process on the active queue starts to execute. When a communication channel becomes ready, the message is passed, and the waiting process is linked to the end of the active process queue. Current process then continues to execute, whenever its turn in the queue comes up.

b. Priority Processes

The transputer T424 supports two levels of priority, high and low priorities. PRIPAR (priority parallel) process may have two components. A queue of active processes is maintained for each priority level. A priority 1 (low priority) process is executed whenever there are no active priority 0 (high priority) processes.

If there are no active priority 0 processes, the latency (that is, the time from an external channel becoming ready, to the start of its first instruction of the relevant waiting priority 0 process) is typically 600 ns. (maximum 2600 ns.). Otherwise, if a priority 0 process is already executing, the relevant waiting process is linked to the end of the priority 0 queue.

c. Performance

The size of a program is given by the sum of the sizes of its program elements. All timing averages and the maximum time to execute the program is given by the sum of the times to execute the individual program elements.

If the program is held in the external memory, the external program fetch time must be added to obtain the program execution time. Because of the instruction lookahaed and the overlap with internal memory, this overhead will usually be small.

If data is held in external memory, the external data access time must also be added to obtain the program execution time. The processor shares memory cycles with its input/output interfaces. Each concurrent access by an interface channel delays the processor by an average of 30 ns. The maximum reduction in performance is 10% ; under typical conditions the reduction is negligible.

2. Memory

4 Kbytes built in static memory provides maximum data transfer rate of 80 Mbytes/sec. The memory interface is a 32 bit multiplexed data and address bus. It extends the internal address capability to a total of 4 GBytes in a single linear address space.

The non-multiplexed cycle provides timing signals to drive industry standard RAM's and ROM's. The multiplexed cycle provides timing signals for RAS[1] and CAS[2] and control for an external address multiplexer. The interface can also provide CAS before RAS refresh cycle.

---

[1]RAS : Row Address Strobe

[2]CAS : Column Address Strobe

An asynchronous wait input is provided so that the memory timing can also be determined externally if required.

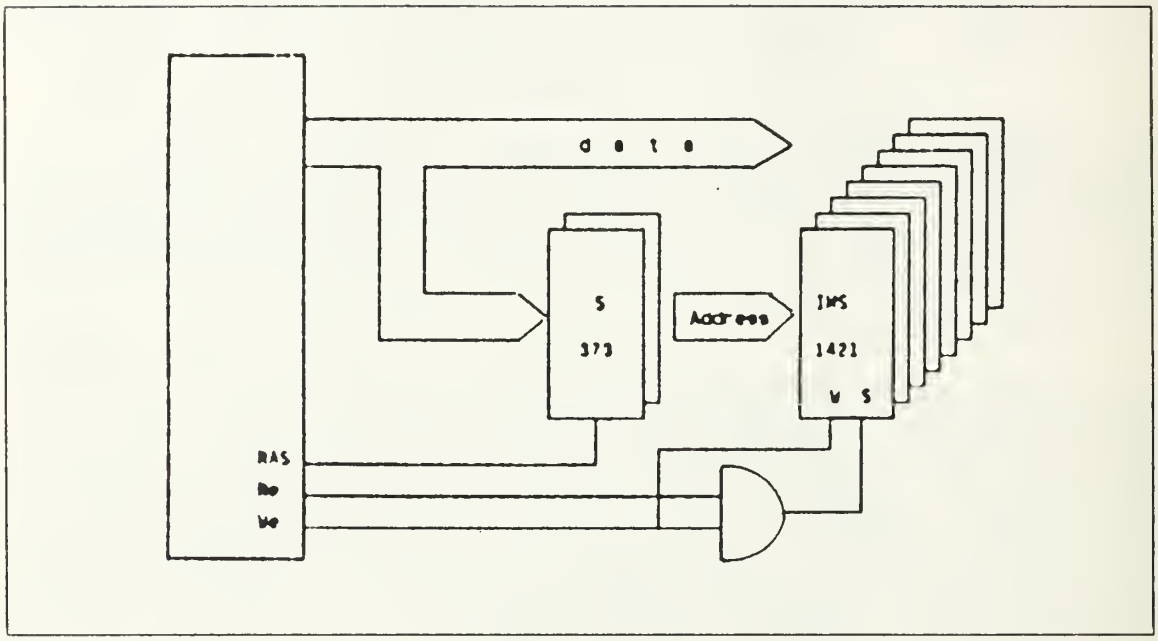

Figure 2.1 Memory Interface Driving Static RAM's

Higher performance can be obtained using static memory. The logic diagram of typical connections to static memory is shown on Figure 2.1 . The cycle to access that memory does not require the phases for address multiplexing and so is completed in 150 nano seconds giving data rate of 25 Mbytes/second maximum.

3. Links

The IMS T424 has four standard INMOS links providing high speed intercommunication between transputer products, enabling rich variety of networks to be constructed. Each link operates independently and provides a memory to memory block message transfer capability between transputers.

Each link implements two OCCAM channels. The link consist of an output and input, both of which are used to carry data link and link control information. A message is transmitted as a sequence of bytes.

After Transmitting a data byte, the sending transputer waits until an acknowledge has been received, signifying that the receiving transputer is ready to receive another byte.

The receiving transputer can transmit an acknowledge as soon as it starts to receive a data byte so that transmission normally continues. This asynchronous protocol guarantees reliable transmission in spite of possible delays in either the sending or receiving transputer.

During transmission of a message , both sending and receiving processes will be set inactive, and they will only be linked to the end of their respective active queues after the final byte has been acknowledged.

The data rate on each link can be programmed, using link set configuration channel [Ref. 4]. The highest frequency is 20 Mbits/sec. giving a maximum data rate of 1.8 MBytes/s. on a channel.

4. Peripheral Interface

The peripheral interface provides access to industry standard devices such as eight bit parallel controllers for auxiliary memory. The interface controller provides a block message transfer capability between memory and the peripheral interface.

The peripheral interface is an 8 bit bidirectional bus which may be used to input and output sequences of bytes. There are two control lines which may be used to address external devices, and an "Event" input to provide an interrupt capability.

The interface is accessed via four standard output channels and four standard input channels. All eight channels use the same 8 bit path and transfer handshake, with the processor initiating the transfer. The transfers are synchronized to a separate external clock, which need not have any fixed relationship with the transputer input clock. Asynchronous operation is also permitted, but at a lower speed than for synchronous operation.

Externally addressable devices may be connected via the peripheral interface. For example, by using one output channel as the address channel, another as the write data channel, and one input channel as the read data channel. Both addresses and data may be arbitrarily long sequences of bytes.

The 4 Mbytes/sec. data rate provided by the interface allows the connection of high performance peripheral chips, without the need for FIFO's or DMA controllers.

The "Event" input may be used to communicate with waiting processes, and hence cause it to be scheduled. This provides an input functionally similar to an interrupt, in a manner consistent with the process model of the transputer. The typical latency for this interrupt is 600 nanosec. The "Event" input can also be used to enable the peripheral interface to respond to being accessed from a standard microprocessor bus.

B. THE TRANSPUTER SYSTEMS CONCEPT

In the past, system performance has increased regularly by a factor of ten each decade, Figure 2.2 . This improvement has been achieved by advances in circuit technology and by increasingly complex systems. For the future, VLSI offers the potential of much greater circuit complexity but only modest increases in circuit performance.

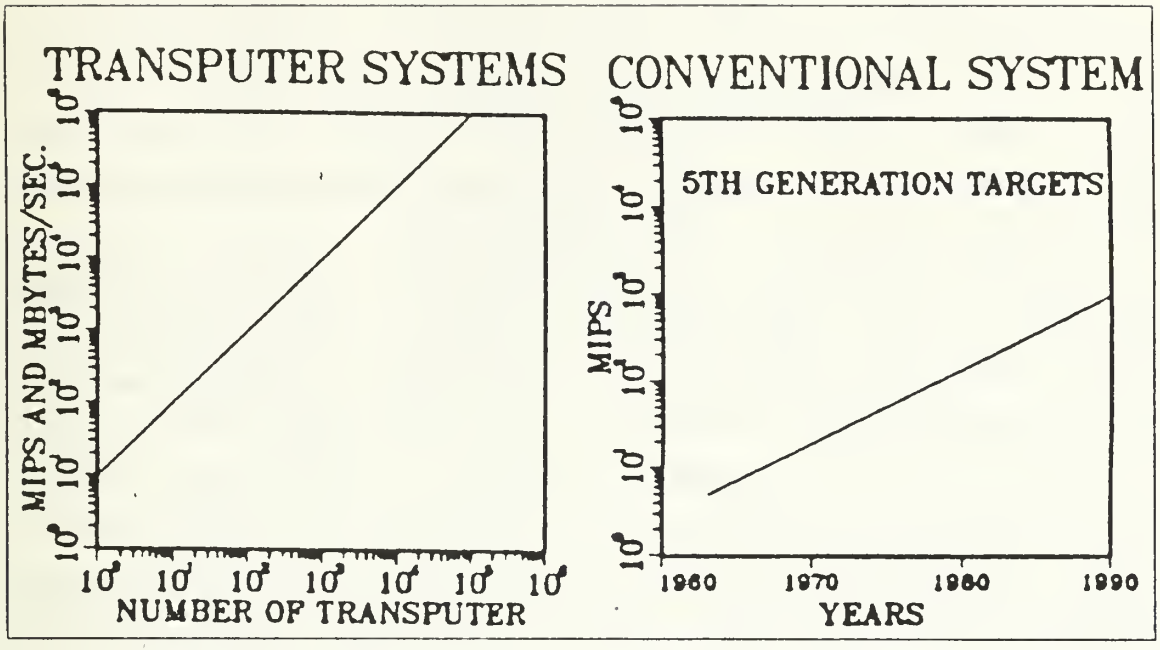

Figure 2.2 Advances in Technology and System Throughputs

The economics of uniprocessor systems are based on the historical perspective that processing is expensive in comparison with the memory. This has led to the Von Neuman bottleneck where a single processor is connected to vast amounts of memory. The economics of the VLSI are different. Today , a single wafer of silicon can contain 2 Mbytes of memory or 256 conventional microprocessors.

To exploit this potential, it will be necessary to build systems with a much higher degree of concurrency than is currently possible. The transputer is designed as a programmable component to implement such systems.

In their proposal to achieve intelligent interaction between people and computers, The Japanese have projected the need for computers with one thousand times the performance of present day systems. These will only be possible using concurrency, and transputer has been designed to make such fifth generation systems a possibility.

Pipelines and arrays of transputers can be used to provide greatly increased performance by exploiting the concurrency inherent in many applications. Two examples which require high performance are signal processing and database searching. Networks of transputers can provide the performance needed for both applications.

Signal processing, such as the fast fourier transform algorithm, maps easly onto a pipeline. The pipeline can accept the input samples at up to 100 KHz., which more than covers the full audio spectrum. A 64 point FFT requires six transputers in the pipeline, a 256 point FFT requires eight and 1024 point FFT requires ten transputers. A pair of pipelines, interlinked at each stage, is able to accept input samples at up to 200 KHz. Higher frequencies can be handled by using more transputers in parallel. [Ref. 6]

A pipeline or an array can also be used to do searching. Provided that the search requests can diffuse through the network, and the answers converge, the shape of the network does not matter - it can even contain faulty devices. The full internal memory of each transputer can be searched 1000 times per second. With external memory attached to each transputer, the search rate is slower, but 64 kbytes per transputer can be searched at least 30 times per second.

Other applications, such as image processing, finite element analysis, matrix manipulation, telephone switching systems , fault tolerant systems and artificial intelligence naturally lend themselves to arrays or networks of transputers. For example the Array structure has been proposed for fault tolerant multi transputer systems , as shown in Figure 4.5 .

III. SOFTWARE CAPABILITIES

OCCAM is a new programming language. It is designed to support concurrent applications in which many parts of a system operate separately and interact. OCCAM is relevant to many present day applications, particularly those involving microprocessors and real time applications. OCCAM will be important for future applications involving the interaction of many thousands of computing components.

The novelty of OCCAM is in its treatment of concurrency. OCCAM enables the programmer to express a program in terms of concurrent processes which communicate by sending messages through communication channels. This has two important consequences. Firstly, it gives the program a clear and a simple structure as the individual processes operate largely independently. Secondly, it allows the program to exploit the performance of many computing components, as each concurrent process may be executed by an individual processor.

OCCAM can capture the hierarchical structure of a system by allowing an interconnected set of processes to be regarded from the outside as a single process. At any level of detail, the programmer is only concerned with a small and manageable set of processes.

A. OCCAM PROCESSES

A process performs a sequence of actions and then terminates. Each action may be an assignment, an input or an output action. An assignment changes the value of a variable, an input receives a value from a channel and an output sends a value to a channel.

At any time between start and termination, a process may be ready and waiting to communicate on one or more of its channels. Communication is synchronous. When both input process and an output process are ready to communicate on the same channel, the value to be transmitted is copied from the output process to input process. The input and output processes then continue.

Each channel provides a one-way connection between two concurrent processes: one of the processes may write to the channel, and the other may read from it.

A process may be ready and waiting to input from any one of a number of channels. In this case, the input is read from the first channel which is used for output by another process.

OCCAM may be used to program a network of computers. Each computer with local store executes a process with local variables, and each connection between two computers implements a channel between two processes.

OCCAM may be used to program an individual computer. The computer shares its time between the concurrent processes, and the channels are implemented by values transmitted in the main memory. Indeed, a program designed for a network of connected computers may also be executed unchanged by a single computer.

B. PRIMITIVES

There are three primitive processes from which all other processes are constructed, as mentioned in the previous paragraph.

These three primitive processes given in Table 3 , can be combined sequentially or concurrently to create more complex processes, and thus they form the building blocks for a program.

TABLE 3

OCCAM PRIMITIVES AND THEIR SYNTAX

PRIMITIVES	SYNTAX
INPUT	channel ? variable
OUTPUT	channel ! variable
ASSIGNMENT	variable := expression

1. Input

An input process reads a value from the channel into a variable. The '?' symbol denotes the input process.

This primitive reads a value from the specified channel. It provides the synchronization with a concurrent process, which outputs a synchronizing signal on the same channel.

An input sets the value of a variable to a value input from a channel. The input waits until an output using the same channel is executed in parallel with the input.

A multiple input is equivalent to a sequence of separate input processes for each variable in turn, in left to right order. Each input is separately synchronized with an output process being executed in parallel. Each variable may be a simple variable, or a word or byte subscripted element of a vector of variables.

2. Output

An output process writes the value of the expression to the channel. The symbol '!' denotes the output primitive.

An output waits until an input using the same channel is executed. It then outputs the value of the

expression to the channel and terminates. A multiple output is equivalent to a sequence of outputs, which writes the value of each expression in turn, in left to right order. Each output is separately synchronized with an input process executed in parallel.

3. Assignment

An assignment process transfers the value of its expression to the named variable.

The expression is evaluated and the variable is set to the resulting value. The assignment process then terminates. The variable may be a simple variable or an element of a vector of variables selected using either byte or word subscription.

C. STRUCTURES

1. Sequential

In many applications it is necessary to do a number of steps one after another, the flow diagram of this structure is given in Figure 3.1 .

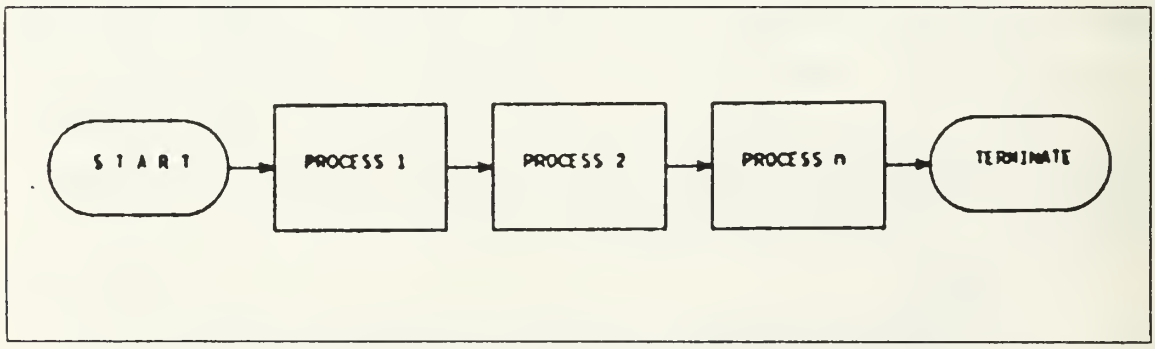

Figure 3.1 Flow Diagram of the Sequential Construct

A sequential process takes the form of the keyword SEQ followed by the component processes, each on a new line, all at an extra level of indentation, as in Table 4 .

TABLE 4
SEQUENTIAL CONSTRUCT IMPLEMENTATION

```
SEQ
  process 1
  process 2
     .
     .
  process n
```

SEQ ensures that each component process terminates before the following component process is executed, and entire process will only terminate after the final component process has finished.

SEQ is an example of an OCCAM constructor. This construct (comprising the SEQ and its component processes, which taken as a whole), can be regarded as a single process.

2. Parallel

If we require many processes to be running as a concurrent system, we can construct a parallel process as seen in Figure 3.2 .

The keyword PAR is followed by a number of component processes, each starting on a new line and indented, as shown in Table 5 . The effect is to execute all of the component processes together, which is achieved by sharing the processor time between the set of active processes.

The parallel construct terminates after all the component processes are terminated. If there is no component process the construct terminates immediately.

Two component processes of a parallel construct may communicate by sending values using a channel. One contains outputs to the channel, and the other contains the inputs from the channel. The processes are said to be connected by the channel. No other component of the parallel construct may use the same channel.

TABLE 5
PARALLEL CONSTRUCT IMPLEMENTATION

```
PAR
  process 1
    -------
    -------
  process n
    -------
    -------
```

Variables are not used for communication between the component processes of a parallel construct. However, a variable may be used in two or more component processes, provided that no component process changes its value by input or assignment.

In addition to the parallel construct, OCCAM contains the prioritized parallel construct declared as PRIPAR. A prioritized parallel construct gives each component process a different priority. The first component has the highest priority and the last component has the lowest priority. An implementation may restrict the number of components which a prioritized parallel construct can have.

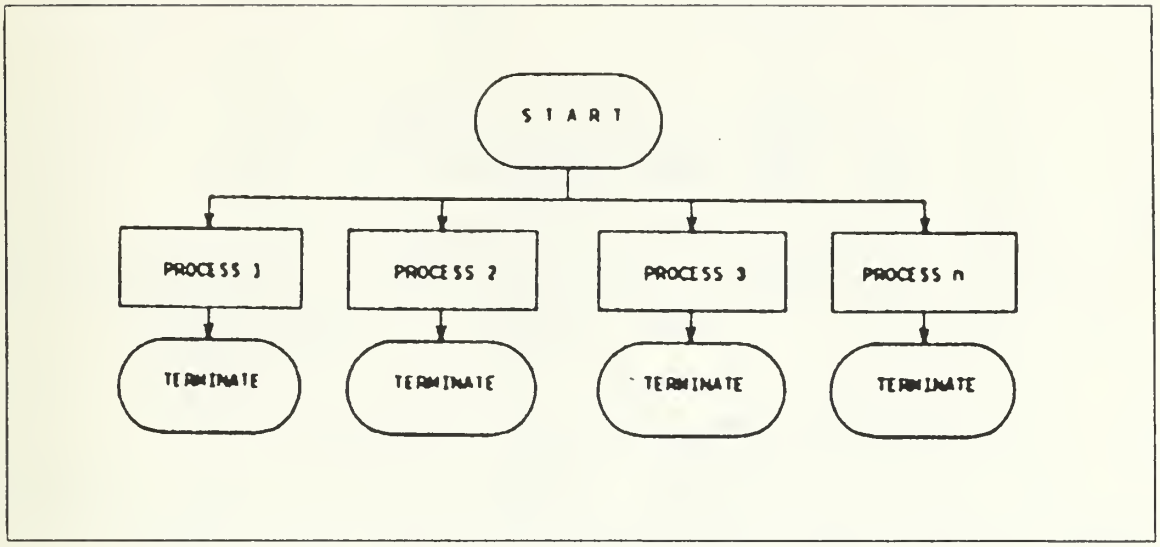

Figure 3.2 Flow Diagram of the Parallel Construct

A prioritized parallel construct ensures that a higher priority process always proceeds in preference to a lower priority one. The progress of a higher priority process is not affected by any lower priority one, except by communication on connecting channels. If several concurrent processes at the same priority are able to proceed, each one is given an opportunity to proceed in turn.

3. Alternative

Sometimes a process has a number of channels associated with it and needs to perform one of a number of actions depending on which channel first sends it a message, Figure 3.3 .

This is achieved using the alternative construct, which chooses just one of its inputs for execution. The keyword ALT is followed by a guarded processes, Table 6 .

An alternative process waits until one of guarded processes is ready to execute. One of the ready guarded

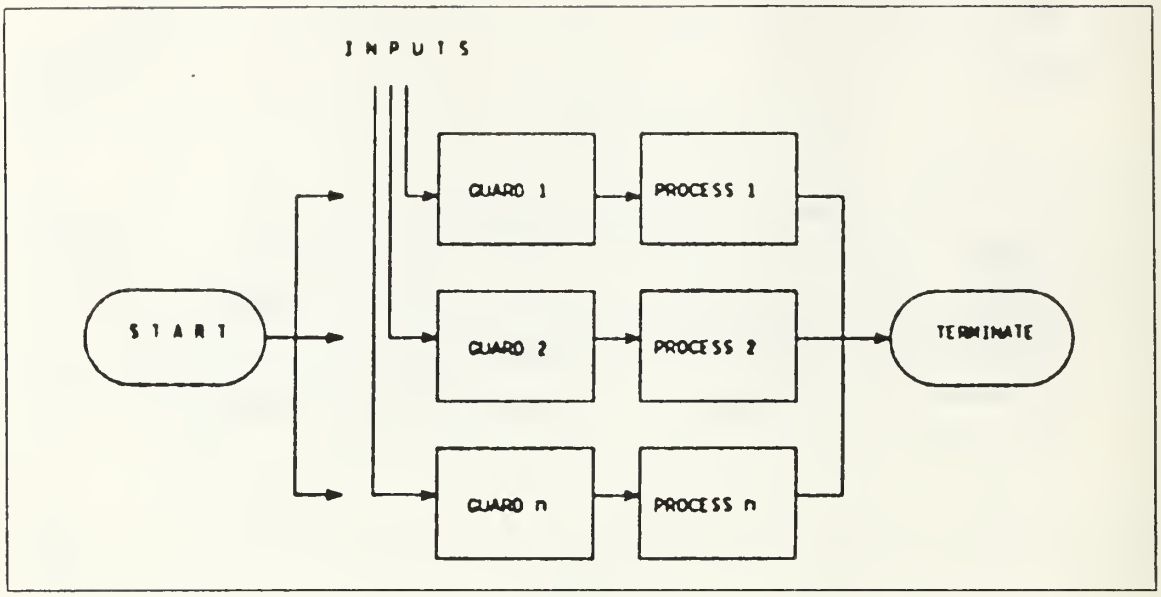

Figure 3.3 Flow Diagram of Alternative Construct

TABLE 6

ALTERNATIVE CONSTRUCT IMPLEMENTATION

```
ALT
  guard-process 1
    process 1
  guard-process 2
    process 2
       .
       .
```

processes is then selected and executed. The construct then terminates. A guarded process starting with an input from a channel is ready if an output process is waiting to output to the channel. If the guarded process is selected, the component process is executed. If a guard contains an expression followed by an input or wait, the guarded process

is ready only if both the value of the expression is TRUE and input or wait is ready. If a guarded process is itself an alternative construct, then it is ready if one or more component guarded processes of the alternative construct is ready. If more than one guarded process becomes ready at the same time, an arbitrary one is selected, this may occur if they contain inputs on the same channel.

4. Repetitive

The repetitive construct takes the form of the keyword WHILE followed by an expression, followed by a single component process indented on the next line, Table 7.

TABLE 7

REPETITIVE CONSTRUCT IMPLEMENTATION

```
WHILE
  expression
    process
```

The component process is executed repeatedly until the expression evaluates to FALSE, and the construct terminates. If the expression is initially FALSE, the process is not executed and the construct terminates immediately, Figure 3.4 .

5. Replicator

A replicator is used with a constructor to replicate the component process a number of times, Table 8 . A replicator can be used with a parallel construct to construct an array of concurrent processes. It also can be used with the

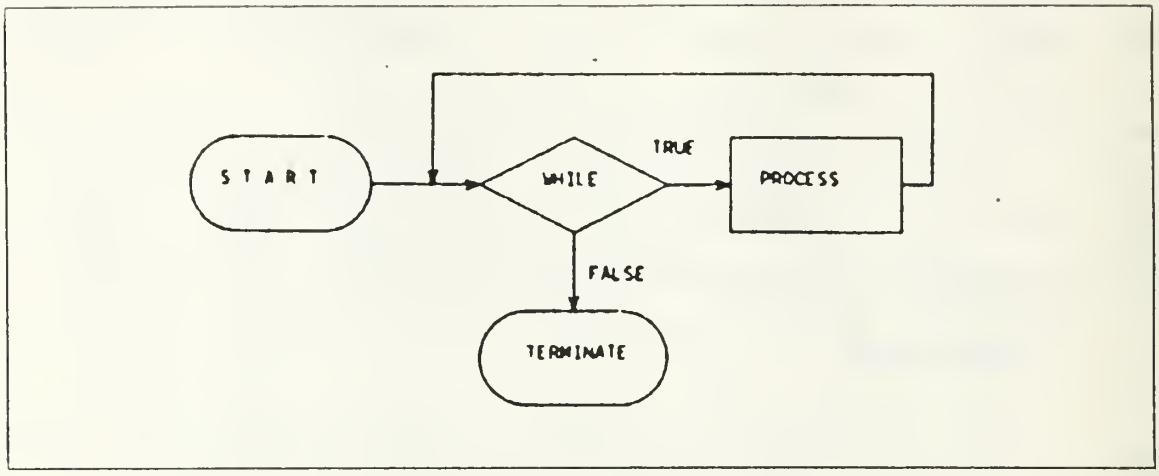

Figure 3.4 Flow Diagram of the Repetitive Construct

alternative construct for reading from an array of channels, and also can be used to provide a conventional loop with sequential construct.

The replicator declares an identifier to be the replicator index, giving its base value and a count of the number of replications required. Its effect is to form a sequential, parallel, alternative or conditional construct containing count components by replicating the component process, substituting successive integer values for the replicator index (starting at base). The substituted value for the replicator index in the last component will be (base + count) - 1.

The replicator index can be used in expressions but not constant expressions, it may not be changed by assignment or input. An implementation may restrict the values of base and count to be constants, particularly when a replicator is used to a form a parallel construct. If a count evaluates less than zero or equal to zero, then an empty construct is generated. This has the effect of termination for sequential, parallel and conditional constructs, and the

TABLE 8

REPLICATOR PROCESS IMPLEMENTATION

```
SEQ  i = [ base  FOR  count ]
  process
```

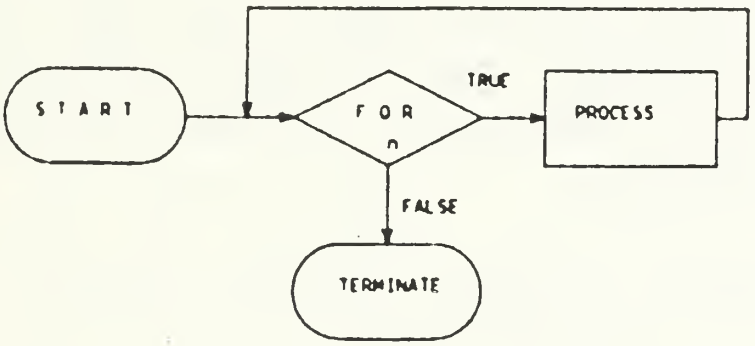

Figure 3.5 Flow Diagram of Replicator Construct

effect of never being ready to execute for alternative processes. The flow diagram of this construct is shown on Figure 3.5 .

6. Conditional

A conditional construct takes the form of an expression followed by a process, and it is able to execute if the expression evaluates to TRUE. The syntax form is shown on Table 9 . A conditional construction takes the form of IF followed by component conditionals which is able to execute if one of its component conditionals is able to execute.

TABLE 9

CONDITIONAL CONSTRUCT IMPLEMENTATION

```
IF
  expression
   process 1
  NOT expression
   process 2
```

The conditional process executes the first component (textually) which is able to execute, and then terminates. If there is no component able to execute, then the construct terminates with no other effect. At most one component is executed, Figure 3.6 .

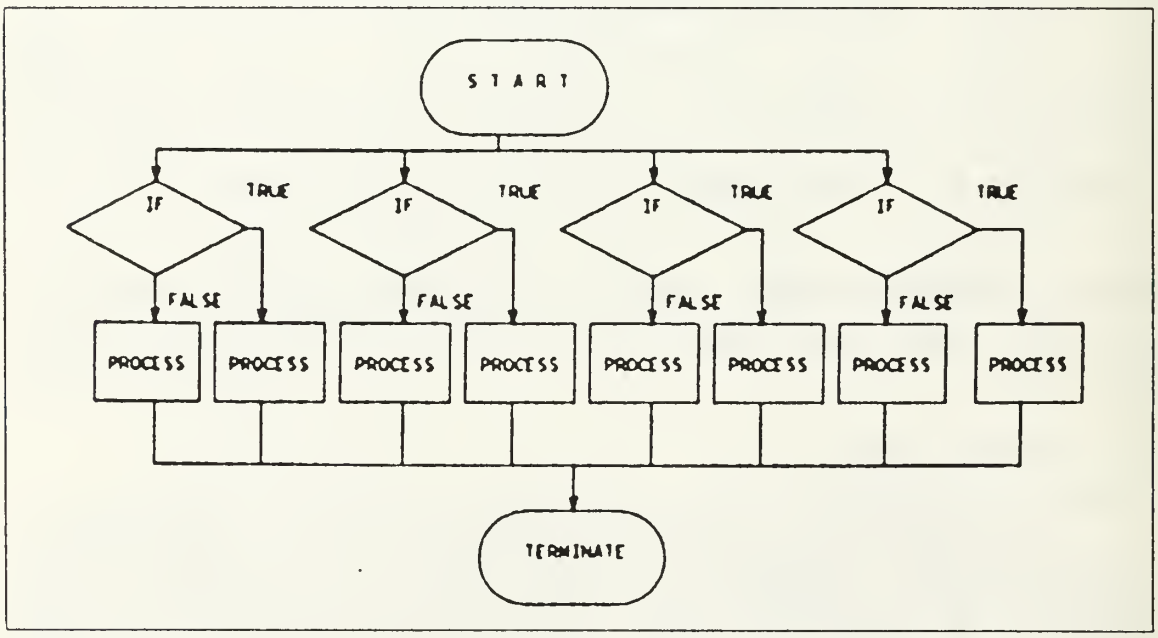

Figure 3.6 Flow Diagram of Conditional Construct

D. CONFIGURATION

Configuration is used to meet speed and response requirements by distributing programs over separate, interconnected computers, and by placing and prioritizing processes on single computers.

Every computer has local store and a set of numbered ports. A physical connection between two computers connects a port on one computer to a port on the other computer. This implements up to two channels between the computers, one in each direction.

A parallel construct may be configured for a network of computers. Each computer executes a component process, and port allocations are used to allocate channels to ports.

A parallel construct may be configured for an individual computer. The computer shares its time between the component processes, and the channels are implemented by values in store. Indeed a parallel construct configured for a network may be reconfigured for an individual computer.

A parallel construct can be used to provide prioritized component processes, and an alternative construct can be used to provide the prioritized input primitives.

The allocation of processing resources to the concurrent processes in a program does not affect the logical behaviour of the program. Simple implementations may omit or ignore some or all the configuration facilities.

IV. SYSTEM STRUCTURE

A. FAULT TOLERANT SYSTEM

1. Objectives

A designer may have multiple objectives for a redundant system. However, central to the theme of fault tolerance are those of,

a. Availability

The average probability that a system is functional at any given time.

b. Error Rate

The average rate at which a system's output makes a transition to an unacceptable state.

c. Reliability

It is described as a mean time to failure for a system. The average length of time a system retains some operational utility without external maintenance.

d. Dependability

It is a statement of system availability through a specified period of time and is a function of availability and reliability.

e. Maintainability

The average period required to return the system to operational status or in some cases to the original state.

f. Growth

The explicit recognition of a system to facilitate realization of future changes.

2. Fundamental Aspects

There are three kind of failures for a system which produce a anomalous condition or function.

Limited "normal" environmental failures that causes components to weaken or break over for a long period of time.

Abnormal environmental failures, disrupting components in a short time or propagating faults introduced from outside of the system.

Man made, hardware or software design flaws, man-machine interaction/interface disruptions or deliberate invasion [Ref. 7].

Malfunctions are anomalous conditions resulting from threats. These include physical malfunctions (component failure), signal or logic level malfunctions (faults), data level malfunctions (errors), control breakdown and system level malfunctions (catastrophic failures).

Fault classification consist of three dimensions:

a. Duration
   Transient, intermittent, pseudo transient, persistent and permanent.

b. Extent
   Local vs. distributed

c. Value
   Determinant or indeterminant.

Temporary failures are the ones most difficult to detect in real time systems.

A transient failure is a nonrecurring temporary failure usually caused by fleeting phenomena such as radiation, noise and power fluctuations.

An intermittent failure is a recurring failure that reappears on a regular basis, often caused by critical tolerances in timing or electrical signal levels. If an intermittent failure is present in a circuit, it may be 'active' at one instant or 'inactive' at another.

Fault tolerant systems resist threats and surmount malfunctions in various ways. Active redundant systems attempt to observe malfunctions and then substitute elements (hardware or software) as required, with a possible interruption in service. Passive redundant systems, often using more resources than active systems, form the correct output based on a consensus of actively redundant elements, with no need to observe the malfunction. Hybrid redundancy combines advantages of active and passive systems by observation and substitution. Self purging systems take advantage of the same techniques through observation and deletion of all failing elements on line. In active redundant systems, substitute elements require testing, else they may be useless when needed. In passive systems, consensus mechanisms are a prime candidate for latent faults.

3. Techniques

The key to successful application of protective redundancy is a systematic and balanced selection of the three forms.

- Hardware (structural) : additional components
- Software (functional) : special programs
- Temporal : Operation repetition

a. Hardware

Methods of introducing hardware redundancy may be divided into two categories, on the basis of terminal activity of the modules.

Static : This method is also known as "masking", since components are employed to mask effects of hardware failures. Three forms of static redundancy have been used in practice; component replication for individual electronic components, triple modular redundancy (TMR) with voting for logic circuits or larger modules of a computer, and quadruple modular redundancy, where four processing elements are used in place of one element. The four elements are paired in two elements per board. The results of each processing step are compared within a board. If the results are found not to compare, the board is declared "faulty", and its results are not permitted to propagate in the system. The board in which the results compare is continuing the operation alone until the faulty board is replaced and normal operation resumed.

Dynamic: In this hardware redundancy approach, fault caused errors are allowed to appear at the terminals of a module. Fault tolerance is then implemented by two consecutive actions. First, the presence of a fault is detected and then a recovery action either eliminates the fault or corrects the error which was caused. Redundancy within the operational system is therefore introduced in a selective rather than massive fashion.

Application of dynamic redundancy requires that a number of design choices be made in the functional design stage.

(1) Modularization. The designer must define a modular architecture with emphasis on a minimal number of connections while trading off desired partitioning.

(2) Fault Detection. The objective is to select real time, concurrent fault detection methods. These methods are error codes, status signals, duplicated operations, internal monitoring of critical events, completion signals, watchdog timers, reasonable checks and totally self checking circuits.

(3) Recovery Actions. Upon detection of the fault, a recovery procedure must take place. If a program restart fails to correct the error, a permanent fault is assumed.

Reconfiguration is a case of replacement in which if a spare is not available, then the processing capability of the machine is decreased, and a decision must be reached concerning: which programs require a lesser degree of survivability, when graceful degradation of processing capacity is acceptable, and a minimum processing capacity is desired prior to entering a safe shutdown mode.

(4) Inter-module Communication Choice. This is a major tradeoff in dynamic systems. Alternatives are bus communication and direct module to module paths.

b. Software Redundancy

Software redundancy includes all additional programs, program segments which may not be included in a computer with fault free hardware. Major forms of software redundancy are.

- Multiple storage of critical programs and data
- n-version programming
- test and diagnosis programs
- Fault tolerant features
- recovery mechanism

Compared to hardware redundancy, one advantage of software redundancy is the ability to superimpose fault tolerance on "off the shelf items". Another advantage is

ease of modification and refinement. The main disadvantage is difficulty of assuring that software will be able to function correctly after a fault occurance or that it will be invoked sufficiently early to prevent system contamination.

c. Time Redundancy

This form of redundancy consist of repeating or acknowledging machine operation at various levels: Microoperations, single instructions, program segments or entire programs. Usually the distinct goals are fault detection by means of repeated execution and recovery by restart or operation reentries.

A common use of time redundancy is found in identification and correction of errors caused by transient faults, and in program restarts after a hardware reconfiguration. This is accomplished by repetition of single instructions, program segments or entire programs.

All these methods may be conveniently grouped according to time of their application with respect to normal system operation.

(1) Initial Testing. Which takes place prior to the normal use and serves to identify elements containing imperfections introduced during production.

(2) Concurrent Detection. Which takes place simultaneously with normal operations. This is implemented by variety of error detecting codes.

(3) Scheduled Detection. Which takes place when the normal operation is temporarily interrupted to test for faults and may be similar to initial testing with the main difference being limited time and a "self test" approach.

(4) Redundancy Testing. Which serves to verify that various forms of redundancy are themselves fault free and ready to act.

B. MULTIPROCESSING CONCEPT

Multiprocessing systems are being, and will be used in a large number of applications such as control of electric power generation, distribution, and consumption, nuclear power processing facilities, safeguarding and control, healthcare delivery in hospitals and medical centers, climate control, security, waste disposal and fire protection in large buildings, and largely in defense systems.

Why are multiprocessor systems useful in all these applications? The reason is several; They usually make it easier for the user to access the system, they generally provide increased performance through resource sharing, and they often increase the availability of a system. A network of microprocessors can quite often duplicate the capability of one large expensive system at lower cost. Multiprocessing systems can provide adaptability and rapid reconfiguration with the system functioning at different times as a very large and complex problem solver or as a network of smaller machines each dedicated to a unique task, or as something between. They can usually also provide increased reliability since the total system can continue to operate despite individual processor failures, albeit with reduced capabilities, provided that some of the links between the processors remain intact. Also, since redundancy can be achieved at a lower cost using processors distributed over a large area, the survivability of the system, particularly in military applications can be increased. Furthermore, a distributed processing system can provide increased, distributed power and responsiveness because it can be closely tailored to the

application. Additional multiprocessor systems can be provided as needed, to ensure proper response time.

Multiprocessor systems can also be designed to be cost effective when applied to a wide variety of applications, where the number of processors can be determined by the distributed processing requirements. A properly designed distributed processing system threatened by overload can be incrementally expanded by simply adding more processors.

The disadvantages may or may not outweigh the advantages, depending on the system-unique requirements. On the minus side, the designer may be faced with increased software complexity. Application software may be more costly to develop for a distributed than centralized system. In contrast to a single central processor based system with only one executive, a distributed system typically requires each processor to contain its own, individual executive that must be capable of communicating with all the other executives in the total system. This, in turn, will require that each individual executive provides a task handling capability where task resident in various processors can communicate with each other, and, in case of local software or hardware errors, diagnostic capabilities exist to localize "bugs". This is not to say that diagnostic or error checking software is not needed or used in large centralized, single processor systems; however, the diagnostic software development for a distributed systems is usually more difficult and costly.

A distributed processing system, by definition, is also more dependent on communication technology, particularly where the computers are widely dispersed and the peak traffic demands between the computers are high.

Finally, the design and development of a unique distributed processors system may require expertise both in hardware and software areas. The advantages and

disadvantages of the distributed multiprocessing systems are given in the Table 10. [Ref. 8]

TABLE 10

ADVANTAGES AND DISADVANTAGES OF MULTIPROCESSING SYSTEMS

ADVANTAGES	DISADVANTAGES
Increased reliability	Increased software
Increased survivability	Difficult system testing
Increased processing power	More communication
Increased modularity	Unique expertise needed
System expandability	

C. DESIGN METHODOLOGY

The purpose of this implementation is to establish a multiprocessing system which provides fault tolerance using a new VLSI product, the T424 transputer microchip. Before the design, the following techniques and methods are assumed for the system.

- Structural hardware redundancy
- Functional software redundancy
- Operation time redundancy

Using more computing elements will provide for both multiprocessing capacity and hardware redundancy. The number of computing elements in the multisystem is chosen as sixteen. The purpose to choose that number for the prototype system is briefly described in the following sections.

Functional software redundancy and operation time redundancy is provided by a fault tolerant operating system design, which is explained later in this chapter.

As described in Chapter II, the hardware provides four communication channels to use in a system configuration, therefore possible system constructions may be listed as following.

a. Pipeline / Ring Structure

This structure consist of each computing element connected to each other with two channels, which also provides redundancy for communication channels between two computing elements, as in Figure 4.1 .

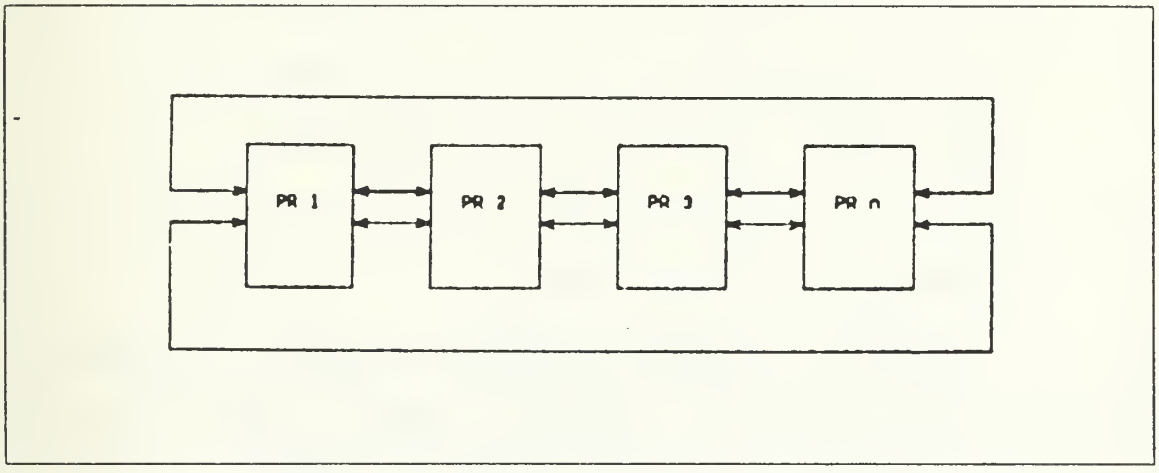

Figure 4.1 Pipeline / Ring Structure

b. Tetragonal 3-D Construction

In this structure each computing element is connected to three elements and they build a new computing group which still has four available communication channels for other computing group connections, Figure 4.2 .

This structure is one of the basic structures that can be found in many kinds higher level of structures also. For example, the matrix structure in Figure 4.4 can

use the tetragonal structure as a computational element which is connected the four neighboring elements.

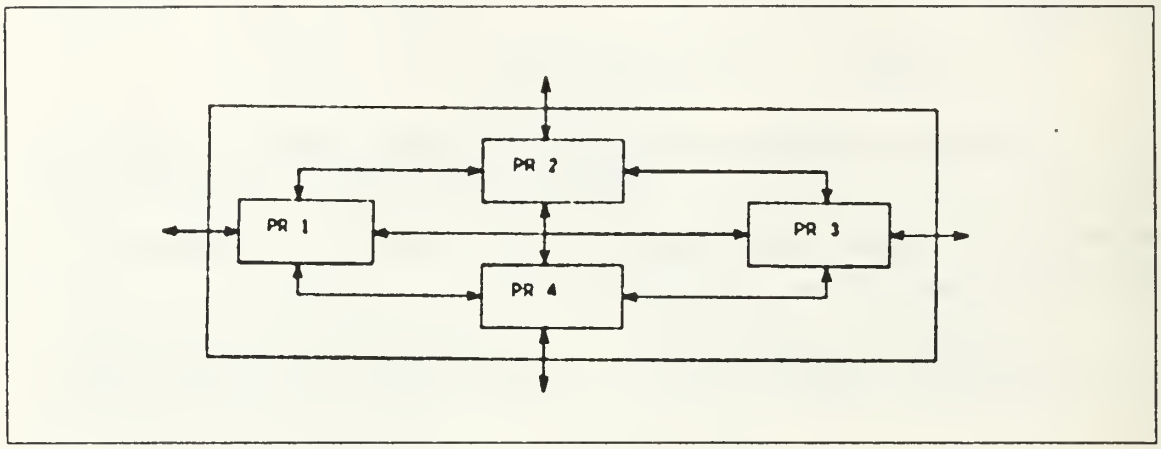

Figure 4.2 Tetragonal 3-D Construction

c. Butterfly Construction

This is a special implementation of a pipeline structure, which is a good solution for the fast fourier transformation or similar engineering applications, Figure 4.3 .

d. Matrix Construction

This structure consist of the connections of the computing elements to each neighboring element in two dimensions. There is no further channel redundancy between two computing element but it provides very large number of communication rings and multiple communication paths. This structure will be described later in this chapter in detail. Basic structure of this type is given in Figure 4.4 .

The matrix structure has been chosen for the proposed prototype multiprocessing system. The basic reason

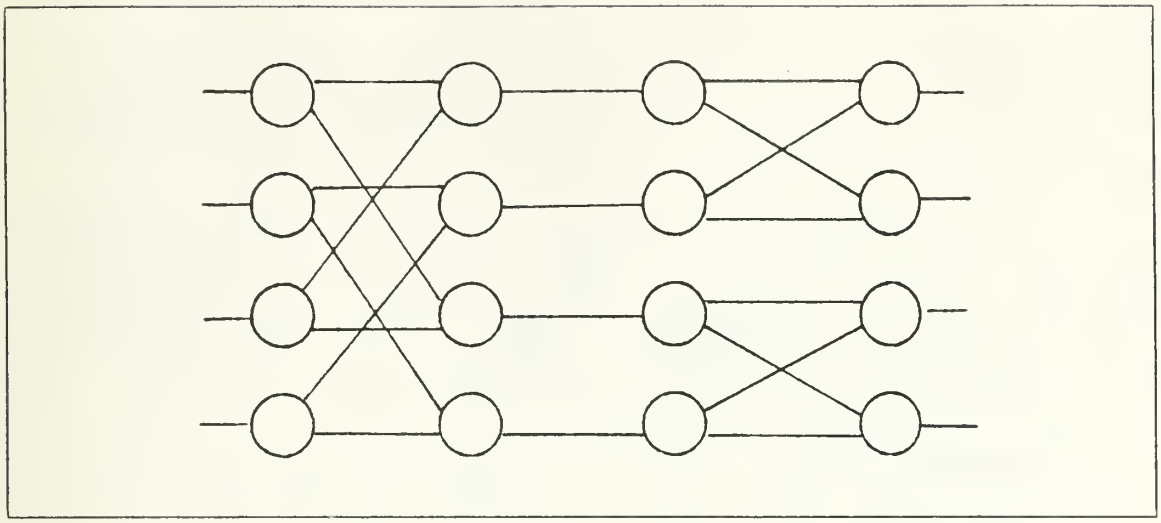

Figure 4.3 Butterfly Construction

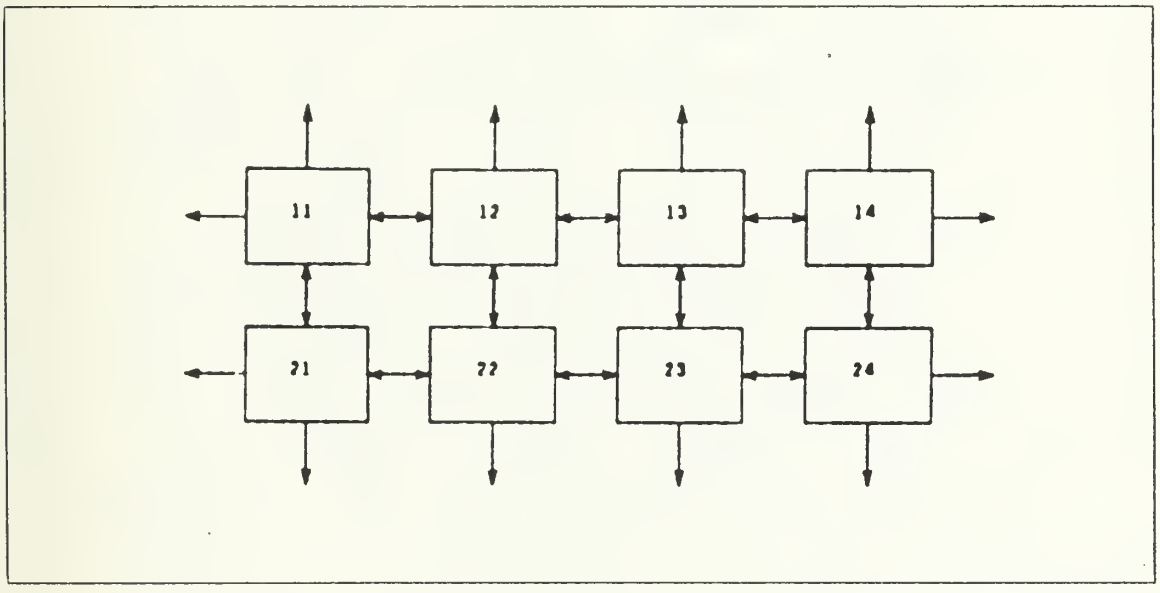

Figure 4.4 Matrix Structure

is that, providing multiple communication paths gives many application orientation possibilities. Many communication rings can be implemented in this kind of system, or pipeline type of processing also can be applied.

The number of processors is determined by the number of rows and columns contained in the matrix. To provide symmetrical structure, the square matrix is a solution. Hence the probable number of computing elements would be 4, 9, 16, 25 and so on.

The number 16 is chosen to build an intermediate prototype multiprocessor. The result of previously discussed design assumptions lead us to the following multitransputer structure in Figure 4.5 .

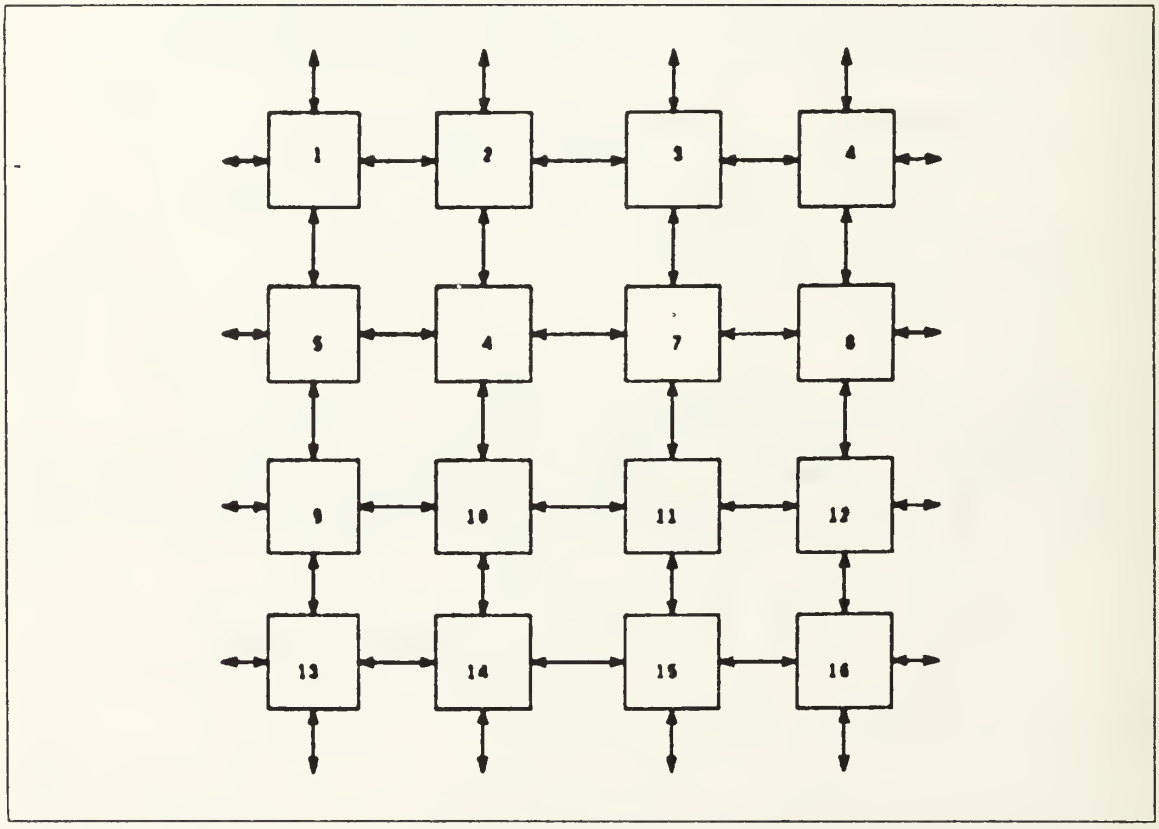

Figure 4.5 The Structure of the Multitransputer System

D. PROPOSED OPERATING SYSTEM

In the previous section the multitransputer systems have been described, and in this section the operating system to be used in the prototype multitransputer system will be described. This operating system includes both multiprocessing and fault tolerance features.

Basically the proposed system includes three main parts: Fault Tolerance Controller, Sequencer and the Link Controllers. These operating system processes work to control the user processes to provide the above mentioned features of the multitransputer system. The prototype operating system structure and its internal connections are shown on Figure 4.6 .

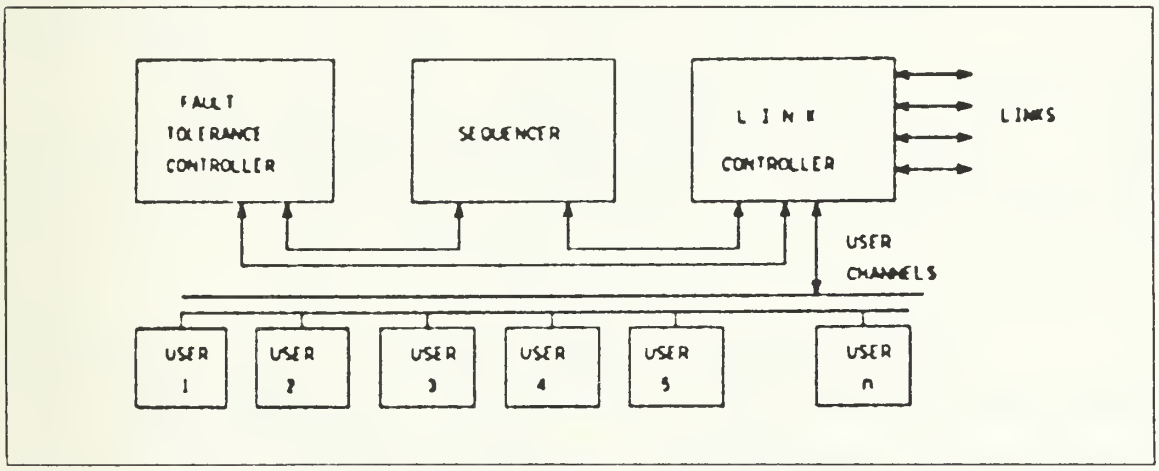

Figure 4.6 Prototype Operating System Structure

a. Fault Tolerance Controller

That subsystem of the operating system includes self check programs, diagnosis programs, watchdog timers, hardware fault tolerance interfaces if provided and others. These processes will be activated by means of fault

detection or run time control.The entire fault tolerance process holds the first priority level among all other subsystems and processes.

The fault tolerance system has not been designed in this thesis, but there will be a high level structural design, a description of its role in the operating system and some suggestions in this section.

The main purpose of this system is the diagnosing and recovering from the errors, which are found both in other systems or in self check processes of the fault tolerance system.

The system inputs come from both link controller subsystem, sequencer subsystem and the fault tolerance system timers or hardware interfaces. The diagnosis or recovery processes must be designed to satisfy the basic objectives of the fault tolerance which are reliability, maintainability and dependability. The system outputs consist of sequencer and link controller outputs, which are used to reconfigure the multiprocessing system and system operation.

b. Sequencer

This subsystem of the operating system provides the multiprocessing system organization. In other words, it determines which particular transputers execute which user processes. Most probably inputs will be both fault tolerance inputs (hardware reconfiguration for particular transputers) and link controller inputs. The link controller provides the communication between other transputers sequencer systems. Therefore the system reconfigurations will be known to the sequencer subsystem.

Probable processes of this system will be some communication protocols for other sequencer subsystems and organization of the processes, such as cancelling some operations or restarting others.

c. Communication Controller

The communication controller subsystem will be explained next in this chapter in detail. The main task of this subsystem is to provide inter and intra communication between processes, systems and transputers. It will also provide a fault detection feature during communications, using watchdog timers and acknowledging techniques.

E. COMMUNICATION SUBSYSTEM DESIGN

1. Design Objectives

As described in the previous paragraph, this subsystem handles the inter and intra communications between processes. These processes could be either user processes, sequencer processes, fault tolerance processes or, if we have, other processes.

Before we explain the design details, it is better to give some conceptual ideas about this system. In the multisystem, Figure 4.5 , transputers are connected to each neighbor in two dimensions, therefore we may use delay insertion loop or token passing type of communication protocols [Ref. 8]. In this design some features of both types of communication has been used to obtain maximum efficiency and redundancy for failures.

A second feature of the communication protocol is to provide error detection, which is achieved using both acknowledging and watchdog timer techniques.

To explain the communication subsystem design, the simple process structure of that system will be explained first. As seen in Figure 4.7 , the communication process accepts inputs from both hardware channels (links) and software channels. These inputs cause the activation of the proper communication process. Execution result will be

another output from either hardware channels (links) or from software channels. This procedure provides the communication between the sender process and the receiver process.

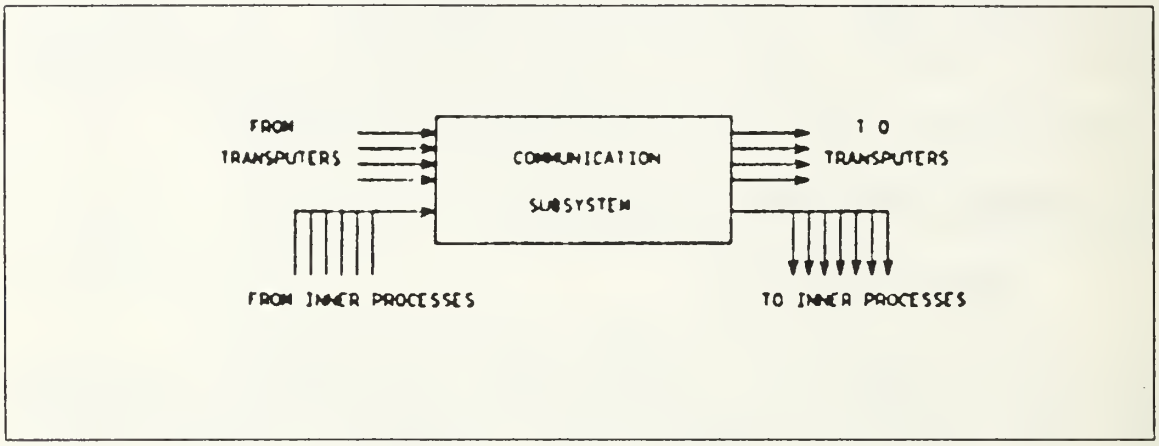

Figure 4.7 Communication System I/O Block Presentation

The communication subsystem has two input groups and two output groups, therefore there will be four different communication types. These are simply described as, from outer transputer to outer transputer (by-pass), from outer transputer to inner process (internal distribution), from inner process to outer transputer (external distribution) and from inner process to inner process (short-cut).

In this communication protocol the token has been used to determine receiver process in the system. The token is the leading byte of the message, which includes the message type, receiver transputer number and receiver channel number information. This token is produced by the communication system itself and also is used by same the system to determine the communication type. The token has been named as CODE for implementation. The CODE is a sixteen bit word two's complement integer in the range -32768 to 32767

After a brief explanation of the subsystem, the design requirements can be listed as follows.

a) Listening to all links and user channels
b) Determination of the communications type
c) Transmission through the proper channel or link

The communication system design is shown in Figure 4.8 , which allows us to achieve the previous design requirements.

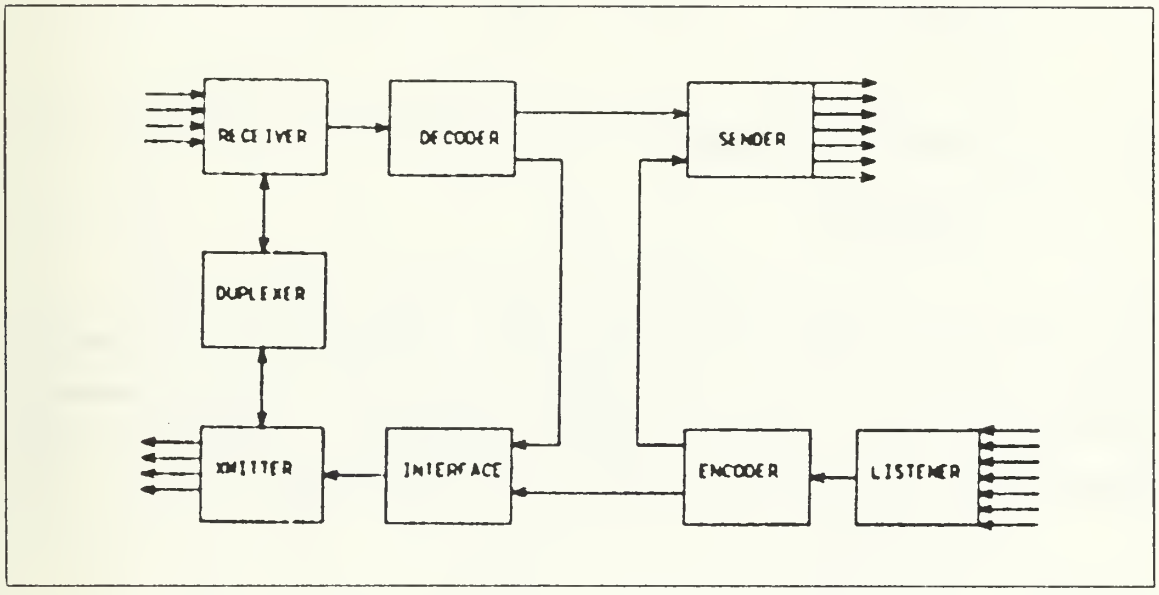

Figure 4.8 Detailed Structure of Communication Subsystem

a. Receiver

Listens to four hardware links and receives both CODE and the following DATA, if it exist, and activates the decoding process. In case of receiving errors, the fault tolerance system is activated.

b. Decoder

Determines if the communication type is a by-pass or an internal distribution, and if it is the internal type, the number of the inner channel is determined.

c. Sender

Sends the activation signal or data taken from both the encoder or decoder to the appropriate process.

d. Listener

Listens to all user channels and receives the data or activation message.

e. Encoder

It is activated by the listening process and determines the CODE according to the multiprocessing configuration provided by the sequencer subsystem. Both transmit or send channels are activated dependant on whether the communication type is distribution or short-cut.

f. Interface

Determines the logical path to send the message to the desired transputer. It uses the hardware configuration which is provided by the fault tolerance subsystem.

g. Transmitter

Transmits the messages through links to other transputers using acknowledgements. Uses the link number determined by its interface process and activates the fault tolerance system in case of transmission problems.

h. Duplexer

Closes the receiver during transmission to prevent the system from false communication.

2. Implementation of the Communication System

Before starting to explain the details of the implementation a brief explanation will be given about CODE (or token) used in this system.

As described before, CODE consist of a two's complement binary sixteen bit word, which can take on the maximum positive decimal value of the 32767. The first digit of the five digit code represent the type of code. In other words, this digit shows whether there is following data or not. If this digit is zero, that means there is no data following the code. If it is 1,2 or 3 data will be assumed to follow.

The next two digits are used to show the process number in case of data not exist. Therefore we can use 99 operating system processes within this system. If the information sequence includes data in that case process number will be determined by first 3 digit, in range of 100 - 327, or we can use 227 user processes within this system.

The last two digits show the transputer number to receive that information sequence. Therefore possible transputer number in this system can be 67 maximum. These values and their meanings are shown on Table 11 .

a. Receiver

The OCCAM program of this subprocess is given in the Appendix A. This subprocess has five inputs and one output. All channels used in this process are bidirectional to provide an acknowledgement procedure. The processing of this subprocess can be separated into three portions.

TABLE 11

CODE AND THE MEANINGS OF THE DIGITS

max.code value	3 2 7 6 7	
without data	0 p p t t	pp : process no tt : transputer no
with data	x q q t t	x : 1,2 or 3 pr.no : (x-1)qq tt : transputer no

The first process is to WAIT for an external message event. This is achieved using alternative structure. All guards are inputs from the hardware links. When an input event occurs, the variable ACTIVE.LINK is assigned to the corresponding link number. This variable is used to make a handshake through the same channel. Also it is used in the interface process to determine the shortest transmission link to reach the destination.

The acknowledgement signal is sent through the active link in the second portion of this subprocess. Also the code type is determined to find out if data exist following the CODE. If data exist, it is received in the same manner as CODE. When the receive process is completed, the decoder process is activated and the receiver process waits for the synchronization signal to prevent the system deadlocks.

Another task of the receiver process can be named as "close receiver". The purpose of this process is to prevent the system from deadlock during the transmission process, because both the receiver and the transmitter processes use the same hardware channels at the same time.

"Close receiver" is achieved by the duplexer subprocess. This process sends the "close receiver" signal through RECEIVER.OFF channel. When this signal is received by receiver, the variable WAIT is assigned as TRUE. This assignment causes waiting for the next synchronization signal from the duplexer, which occurs at the end of transmission procedure. Wait function will be disabled, before the program starts to wait for the external input through the links.

b. Decoder

This subprocess has one input and two outputs. It is activated by the receiver process and the transputer number of the code is determined by looking at the last two digits. The next step is to decide if the communication type is internal or by-pass. According to the communication type either the sender process or the transmit interface process is activated. After each activation, the process waits for a synchronization signal from the activated process. The last step before the termination of the process is to send a synchronization signal to the receiver process. The OCCAM program implementation of this process is given in Appendix-A.

c. Sender

This subprocess is activated either by the decoder process or the encoder process. The only task of this process is to send the message to the specified user process (or operating system process). Before the termination of the process, a synchronization signal is returned to the invoking process.

d. Listener

This process consists of repetitive alternative constructs and guarded processes which are reading from the user and/or operating system processes. When the input is taken from any one of these channels, the encoder process is activated to continue the communication procedure. The listener process waits for acknowledgement from the encoder process to terminate itself.

e. Encoder

This subprocess is activated by the listener subprocess. The first procedure is to determine the receiving transputer number. This is achieved using the PROCESS.TABLE provided by the sequencer subsystem.

The next procedure is to determine the communication type (either external distribution or short-cut). This is achieved by comparing the target transputer number with the number of the "own" transputer.

In case of a short-cut, the send subprocess is activated to send the message to an internal process. If the message is targeted to an external transputer's process, then a code word is computed for either with data or without data cases, and the transmit interface subprocess is activated to send that information through the proper hardware link.

The synchronization signal is waited from the interface process before the termination. After that signal is received, the listener process is released by synchronization signal and process terminates.

f. Interface

This subprocess is activated either from the decoder or the encoder depending on the communication type.

The main task of this subprocess is to determine the logical and the shortest link number to reach the destination. This is achieved using distance tables and the hardware status table. The distance tables for the first, second and third priority levels are given in the Appendix C. The basic idea is to compare hardware status of the chosen link to reach the specified transputer and continue the comparison to find the available link. If the communication type is a by-pass, then the link number must be chosen differently from the active link. In other words, we can not send the same message back through the same channel again, which causes system deadlock.

After finding the proper link number, the transmitter subprocess is activated to transmit the message. The acknowledgement must be received before the synchronization signal is sent to the encoder or the decoder subprocesses and the interface process terminates.

g. Transmitter

This subprocess is activated by the interface subprocess and after receiving the necessary parameters for transmission, the duplexer subprocess is activated to terminate the receiver subprocess to prevent ourselves from a deadlock and faulty communication. When the duplexer subprocess gives the "receiver is closed" signal, the transmit interface subprocess is released by the synchronization signal. The transmission procedure continues in the same manner as the receiving process. After transmission is completed, the duplexer unit is activated to release the receiver process, and transmit process terminates.

h. Duplexer

As mentioned before, this subprocess provides closing and opening of the receiver during the transmission

cycle. First it waits for the event message from the transmitter to send a close message to the receiver. Then it acknowledges the transmitter when the "receiver closed" message is received from the receiver subprocess. After that the duplexer waits to get "transmission completed" message from the transmitter process and sends the "open receiver" message to the receiver process before the termination of the process.

V. PERFORMANCE EVALUATION

In this chapter the performance of the communication system will be evaluated. The first step is the calculation of the subprocesses execution times for different cases.For example, the execution time of the sender process will be calculated both for the decoder activation and the encoder activation separately. This calculation will allow us to calculate the total execution times of the different types communications.

A. SUBPROCESSES EXECUTION TIMES

Table 2 is used to evaluate the execution times of the processes. For a particular system application, IMS 1400 static RAM and IMS 3630 erasable ROM are assumed as external memories. Therefore, in order to calculate additional external program and data access times, the access times of these memories are used [Ref. 4].

As an example, the receiver process execution time will be calculated. The method that has been used to calculate the execution time is taken from [Ref. 6]. The calculation table includes every type of construct, evaluation and operand in the OCCAM. By inspection of the receiver program in Figure 5.1 , the calculation table can be filled as follows.

The receive process contains the alternative construct with 5 guard processes, 5 conditional branches, 1 replicative construct and no parallel construct.

During the execution there will be 2 parenthesis, 11 constant, 13 variable and 10 vector variable evaluations.

The receive process executes 5 input and 3 output primitives (maximum case). The number of arithmetic operations are 2 division, 2 comparison and 2 logic statements.

```
PROC receive (CHAN decode.advance, receive.off) =
  VAR acknowledge, wait :
  WHILE TRUE
    SEQ
      wait := FALSE
      ALT
        link[1] ? ext.code
          active.link := 1
        link[2] ? ext.code
          active.link := 2
        link[3] ? ext.code
          active.link := 3
        link[4] ? ext.code
          active.link := 4
        receive.off ? ANY
          wait := TRUE
      IF
        NOT wait
          SEQ
            link[active.link] ! ext.code
            link[active.link] ? acknowledge
            ext.code.type := FALSE
            IF
              acknowledge
                IF
                  (ext.code/10000) = 0
                    ext.code.type := TRUE
                  (ext.code/10000) <> 0
                    SEQ
                      link[active.link] ? ext.data
                      link[active.link] ! ext.data
                      link[active.link] ? acknowledge
              TRUE
                SKIP
            IF
              acknowledge
                SEQ
                  decode.advance ! ANY
                  decode.advance ? ANY
              NOT acknowledge
                fault.links[rcv.trouble] ! active.link
```

Figure 5.1 The Receiving Process

These counted values provide the quantities for the calculation table. Subtotals are found by multiplying the quantities with their execution times, and the summation of

these subtotals gives us the total execution time of the receiver process.

In the following section these calculations will be used to estimate the worst case communication performances. Some of these results may have larger values, but actual execution times will not be greater than calculated execution times.

The following thirteen tables show the instruction types of the processes and their execution time calculations. In these calculations the worst cases are assumed.

TABLE 12

RECEIVE PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600n	TOTAL
QUANTITY	5	5	-	1	
SUBTOTAL	6940	2750	0	600	10290
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY	2	11	13	10	
SUBTOTAL	156	1562	4082	4090	9890
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY	5		3		
SUBTOTAL	6450		3870		10320

OPERANDS	+,-	/ , \	*	COMPARE	LOGIC	
	105	1950	950	210	105	
QUANTITY	-	2	-	2	2	
SUBTOTAL	0	3900	0	420	210	4530
TOTAL (nano second)						35030

TABLE 13

RECEIVER CLOSING PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY SUBTOTAL	5 6940	2 1100	- 0	1 600	8640
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY SUBTOTAL	- 0	5 710	3 942	- 0	1652
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY SUBTOTAL	2 2580		1 1290		3870

OPERANDS	+,- 105	/ , \ 1950	* 950	COMPARE 210	LOGIC 105	
QUANTITY SUBTOTAL	- 0	- 0	- 0	- 0	1 105	105
TOTAL (nano second)						18032

TABLE 14

DUPLEXING PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY	-	-	-	1	
SUBTOTAL	0	0	0	600	600
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY	-	-	6	-	
SUBTOTAL	0	0	1884	0	1884
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY	3		3		
SUBTOTAL	3870		3870		7740

OPERANDS	+,-	/ , \	*	COMPARE	LOGIC	
	105	1950	950	210	105	
QUANTITY	-	-	-	-	-	
SUBTOTAL	0	0	0	0	0	0
TOTAL (nano second)						10224

TABLE 15

LISTENING PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY	376	-	-	1	
SUBTOTAL	377940	0	0	600	378540
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY	-	-	3	1	
SUBTOTAL	0	0	942	409	1351
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY	2		1		
SUBTOTAL	2580		1290		3870

OPERANDS	+,-	/ , \	*	COMPARE	LOGIC	
	105	1950	950	210	105	
QUANTITY	-	-	-	-	-	
SUBTOTAL	0	0	0	0	0	0
TOTAL (nano second)						383761

TABLE 16

DECODING TO SEND PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY SUBTOTAL	- 0	1 550	- 0	1 600	1150
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY SUBTOTAL	2 156	2 284	10 3140	- 0	3580
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY SUBTOTAL	2 2580		2 2580		5160

OPERANDS	+,- 105	/ , \ 1950	* 950	COMPARE 210	LOGIC 105	
QUANTITY SUBTOTAL	- 0	2 3900	- 0	1 210	- 0	4110
TOTAL (nano second)						14000

TABLE 17

DECODING TO XMIT PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY	-	2	-	1	
SUBTOTAL	0	1100	0	600	1700
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY	1	1	10	-	
SUBTOTAL	78	142	3140	0	3360
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY	2		2		
SUBTOTAL	2580		2580		5160

OPERANDS	+,-	/ , \	*	COMPARE	LOGIC	
	105	1950	950	210	105	
QUANTITY	-	1	-	2	-	
SUBTOTAL	0	1950	0	420	0	2370
TOTAL (nano second)						12590

TABLE 18

SENDING BY DECODER PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY SUBTOTAL	2 3940	2 1100	- 0	1 600	5640
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY SUBTOTAL	- 0	- 0	5 1570	1 409	1979
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY SUBTOTAL	1 1290		2 2580		3870

OPERANDS	+,- 105	/ , \ 1950	* 950	COMPARE 210	LOGIC 105	
QUANTITY SUBTOTAL	- 0	- 0	- 0	- 0	1 105	105
TOTAL (nano second)						11594

TABLE 19

SENDING BY ENCODER PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY	2	2	-	1	
SUBTOTAL	3940	1100	0	600	5640
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY	-	-	5	1	
SUBTOTAL	0	0	1570	409	1979
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY	1		2		
SUBTOTAL	1290		2580		3870

OPERANDS	+,-	/ , \	*	COMPARE	LOGIC	
	105	1950	950	210	105	
QUANTITY	-	-	-	-	1	
SUBTOTAL	0	0	0	0	105	105
TOTAL (nano second)						11594

TABLE 20

ENCODING FOR SEND PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY SUBTOTAL	- 0	1 550	- 0	1 600	1150
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY SUBTOTAL	- 0	- 0	8 2512	1 409	2921
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY SUBTOTAL	2 2580		2 2580		5160

OPERANDS	+,- 105	/ , \ 1950	* 950	COMPARE 210	LOGIC 105	
QUANTITY SUBTOTAL	- 0	- 0	- 0	1 210	- 0	210
TOTAL (nano second)						9441

TABLE 21

ENCODING FOR XMIT PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY SUBTOTAL	- 0	4 2200	- 0	1 600	2800
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY SUBTOTAL	1 78	4 568	16 5024	1 409	6079
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY SUBTOTAL	2 2580		2 2580		5160

OPERANDS	+,- 105	/ , \ 1950	* 950	COMPARE 210	LOGIC 105	
QUANTITY SUBTOTAL	1 105	- 0	1 950	4 840	1 105	2000
TOTAL (nano second)						16039

TABLE 22

XMIT PREPARATION BY DECODER PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY SUBTOTAL	2 3940	6 3300	- 0	1 600	7840
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY SUBTOTAL	6 468	3 426	26 8164	6 2454	11512
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY SUBTOTAL	2 2580		2 2580		5160

OPERANDS	+,- 105	/ , \ 1950	* 950	COMPARE 210	LOGIC 105	
QUANTITY SUBTOTAL	- 0	- 0	- 0	- -	4 420	420
TOTAL (nano second)						24932

TABLE 23

XMIT PREPARATION BY ENCODER PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY SUBTOTAL	2 3940	3 1650	- 0	1 600	6190
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY SUBTOTAL	- 0	- 0	20 6280	7 2863	9143
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY SUBTOTAL	2 2580		2 2580		5160

OPERANDS	+,- 105	/ , \ 1950	* 950	COMPARE 210	LOGIC 105	
QUANTITY SUBTOTAL	- 0	- 0	- 0	- 0	1 105	105
TOTAL (nano second)						20598

TABLE 24

TRANSMITTING PROCESS EXECUTION TIME

CONSTRUCT	ALT. 1000n+1940	COND. 550 n	PARALLEL 1435n-100	REP. 600	TOTAL
QUANTITY SUBTOTAL	- 0	6 3300	- 0	1 600	3900
EVALUATION	PARENT. 78	CONSTANT 142	VARIABLE 314	VECTOR 409	
QUANTITY SUBTOTAL	- 0	2 284	28 8792	5 2045	11121
PRIMITIVE	I N P U T 1290		O U T P U T 1290		
QUANTITY SUBTOTAL	4 5160		5 6450		11610

OPERANDS	+,- 105	/ , \ 1950	* 950	COMPARE 210	LOGIC 105	
QUANTITY SUBTOTAL	- 0	- 0	- 0	3 630	2 210	840
TOTAL (nano second)						27471

B. SERIAL COMMUNICATION PERFORMANCE

As described in Chapter IV, there are four different types of communication. The execution times of these communication types will be calculated separately in this section. These four communication types and their subprocesses are listed in Table 25 .

TABLE 25
COMMUNICATION TYPES AND INVOKED PROCESSES

COMM. TYPE	PROCESSING SEQUENCE
By-pass	Receive-decoder-Interface-duplexer-transmitter
Internal Dist.	Receive-decoder-send
External Dist.	Listen-Encoder-interface duplexer-transmitter
Short-cut	Listen-Encoder-send

1. By-Pass Communication Performance

In this communication the invoked subprocesses are receive, decoder, interface (activated by decoder), transmitter and duplexer. Using Table 12, 13, 14, 17, 22 and 24, the total time for the by-pass process is found to be 128.5 microseconds.

2. Internal Distribution Performance

In this type of communication the receive, decoder, send (activated by decoder) processes are executed. Using

Table 12, 16 and 18, the total execution time of this type of communication is found to be 60.6 microsecond.

3. External Distribution Performance

In this kind of communication, the invoked subprocesses are listen, encoder, interface (activated by encoder), transmit, duplexer and receiver closing. Using Table 13, 14 15, 21, 23 and 24, the total execution time of the external distribution type communication is found to be 447.8 microsecond.

4. Short-Cut Procedure Performance

Short-cut type of communication consist of listen, encoder and send (activated by encoder) processes. Using Tables 15, 19 and 20, the execution time of the short-cut communication is found to be 404.8 microsecond.

The last two types of communication's execution times are found to be about 400 microseconds. The reason for this time is the large number of processes, which are monitored by the communication system. This execution times can be reduced to 50 - 100 microseconds by using a smaller number of user processes. Also operating system processes can be monitored separately than user processes.

C. FAULT FREE SYSTEM PERFORMANCE

In a system with parallel bus and common memory , as in Figure 5.2 , the block data transfer performance will be calculated in this section.

That kind a operation requires bus check routines and the data transfer protocols. The bus checking procedure execution time can be neglected comparing to the data transfer time, therefore the problem is simplified.

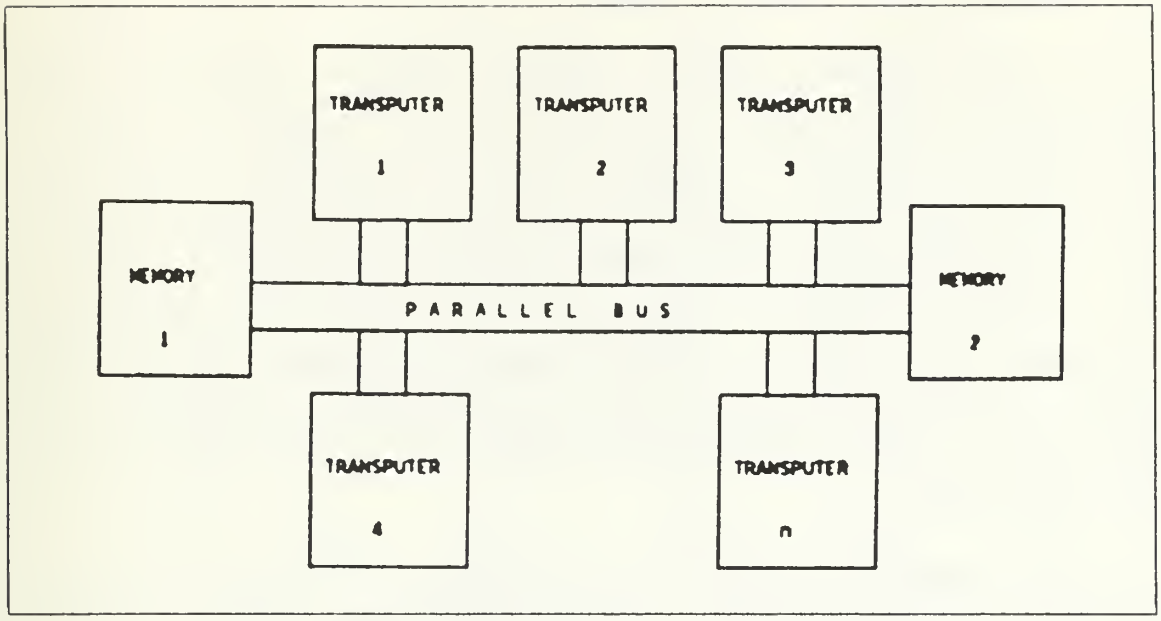

Figure 5.2 Fault Free System Structure

1000 * 4 Bytes of data transfer requires 1000 memory access cycles. In this procedure there is no communication or fault tolerance control procedures, therefore the minimum time to access these common and local memories, which are the same kind of static memories, is calculated using Table 2 as follows,

identifier evaluation =	120 + 1.7 * 55	= 213.5
memory access		= 100.0
TOTAL		313.5

4 KBytes of information requires 313.5 microsecond transfer time, or approximately the data transfer rate for parallel bus structure is approximately 100 Megabits per second.

D. SYSTEM WITHOUT SHARED MEMORY

In the system without shared memory (parallel bus failure mode) the common data base will be assumed in one of the resident transputer's memories. To transfer this data to other transputers requires at least a four stage operation, where each stage has been denoted with stars in Figure 5.3 . In each stage a regular communication protocol will be assumed. The special subroutines can be provided for this purpose, therefore the total data transfer rate will be dependent on the serial data transfer rate and communication execution times.

Without fault tolerance controlling, the data transfer procedure execution time for 4 * 1000 bytes will be determined by local memories access times and serial data transfer rates, as follows;

```
i/o primitives   :  50 * 1000 + 625
serial transfer  : 625 * 1000
memory access    : 100 * 1000 + 500
TOTAL . . . . = 776290 nano sec.
```

Distribution of 4000 Bytes of data to all transputers will require the execution of the same procedure four times. Therefore total time consumed at data transfer will be 3105.1 microseconds. This corresponds to 3.1 microseconds per byte or approximately 0.3 Megabytes per second data transfer rate. This evaluation is true, if there is no interference in the internal buses.

If the interference is assumed on internal buses, in that case there will be delays at each stage of data transfer. These delays cause to reduction of the data transfer rate.

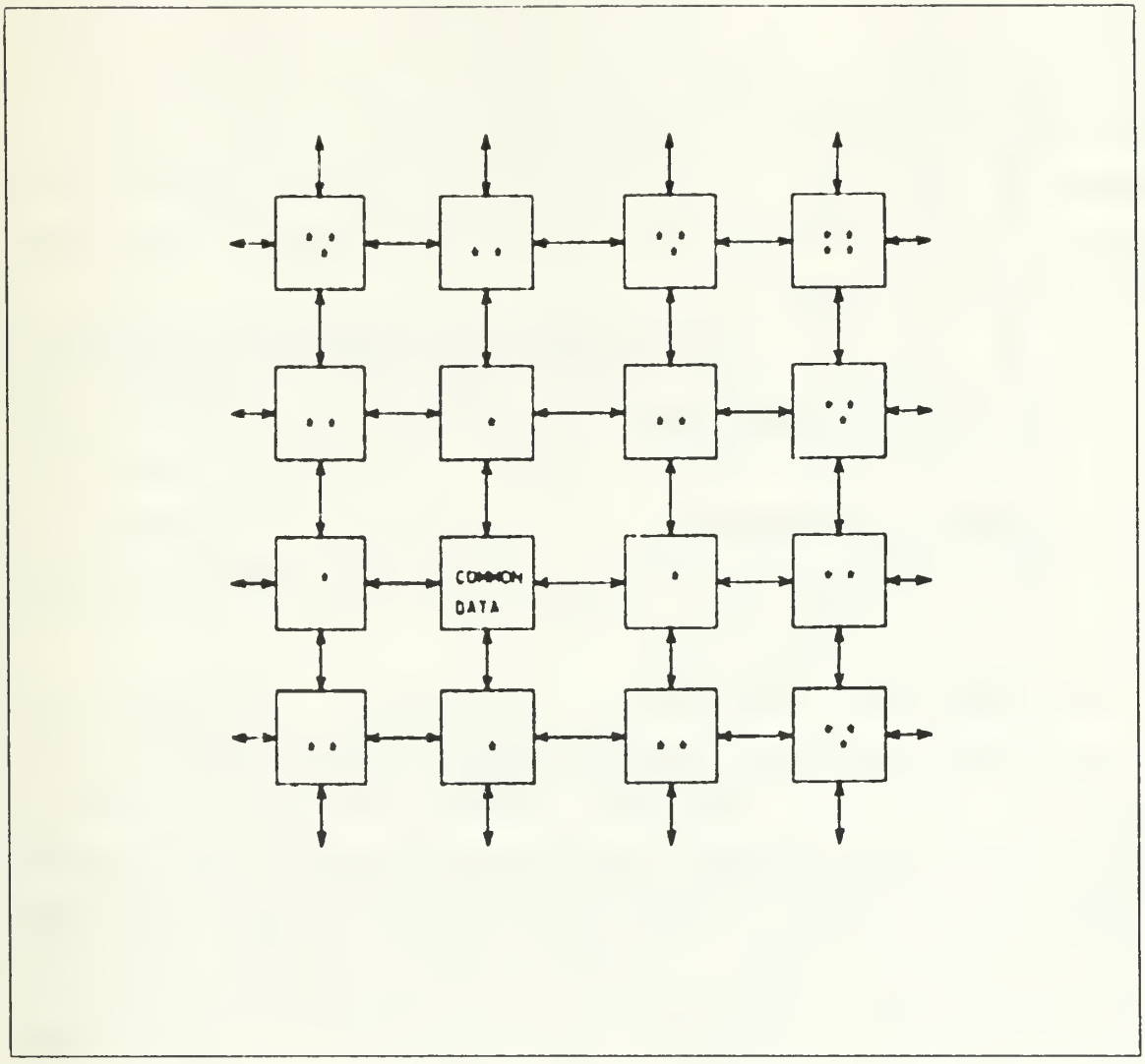

Figure 5.3 Common Data Transfer

E. EXPANDABILITY

In this section different computation models will be tried to analyze the performance of the multitransputer system, and to show the system expandability.

Assume that the computation proceeds in a linear pipeline fashion. This model can be adapted into the multitransputer system in many different configurations. To provide triple or quadruple modular redundancy, a

multitransputer system may include three or four computation models in itself. The basic reason is to provide simultaneous throughputs, and by comparison it will be possible to find hardware faults, if comparisons fail. Using triple or quadruple redundancy, hardware failures will be detected and located, also correct resultS will be forwarded to the next stage of calculation.

As an example of triple modular redundancy, a linear type of computation model can be used. If we assume that each row of a multitransputer system corresponds to a linear type computation model, and the first three rows execute the same program simultaneously. If one of the three results is different, this row's computing elements are assumed failed. This row's execution job is assigned to a spare row while the failed row is diagnosed.

For quadruple redundancy, the system's components are arranged into removable units in such a fashion that on each removable unit two computing elements constantly produce resultss which are checked for consistency. If the two units generate inconsistent results, the removable unit is assumed to have failed and it will be removed from the system by service personell. A spare unit is placed into the system while the other working pair of computational units has propagated correct results. To allow the spare unit to be switched into the system without powering the system down is an important design feature included in such systems. Figure 5.4 shows an example of this computation model and its adaptation.

The output element of each line carries out the comparison procedure. These elements are denoted by "*" in the Figure 5.4 .

This type of a pipeline computation can be used in the real time systems to compute some time dependent events. For

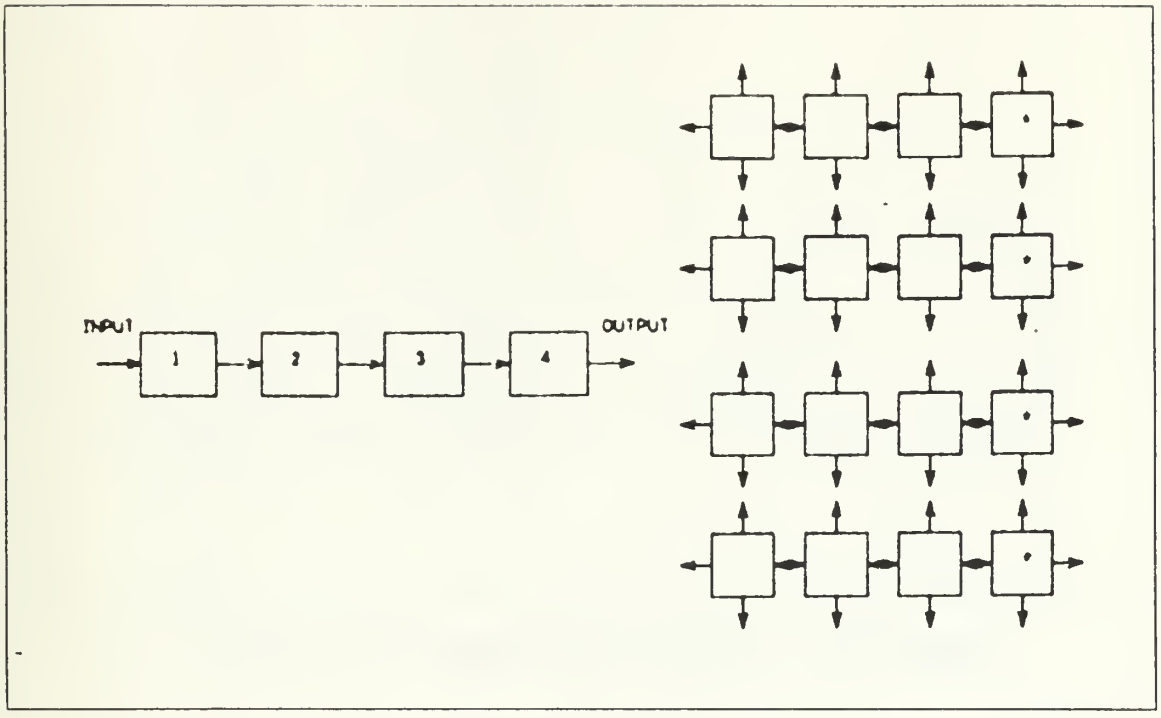

Figure 5.4 Linear Type Computation Model

example radar information would be input, second and third computing elements can calculate the actual target coordinates and weapon system outputs. These values can be compared by the last elements of the model to decide correct results.

Another computation model could be a loop type of computation. Also triple or quadruple modular redundancy can be applied to that model using three or four computation loops in the multitransputer system. The computation output elements must be assigned as decision elements of the redundant system, as in Figure 5.5

This type of computational model can also be used in recursive calculations such as discrete signal analysis. Also triple or quadruple voting process will provide correct results at the end of each discrete period.

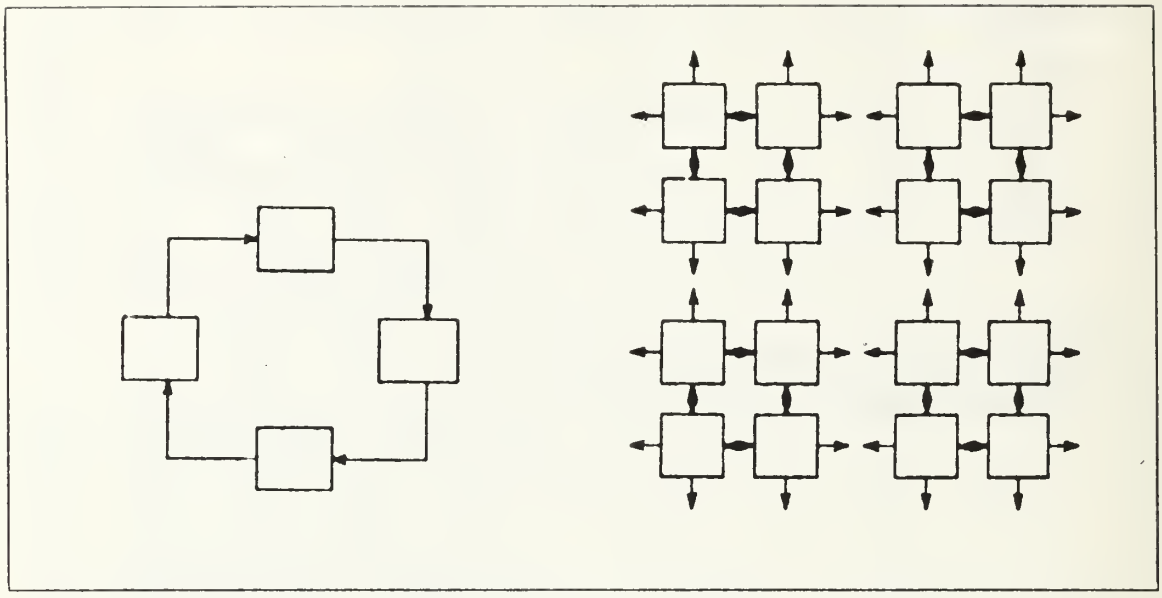

Figure 5.5 Loop Type Computation Model

A star type of computational model can be applied into the multitransputer system also. As in previous applications, triple or quad uple modular redundancy is provided by three or four computation groups in the system. An example of that kind a system is shown on Figure 5.6 .

That type of modular design may be used to compute three stage computations. For example the first two computing elements can be used to evaluate both search and track radar input for a certain time period and in the next stage target evaluations can be done, and final stage can be used to demonstrate these computations.

Another combination of modular designs also can be applied to the multitransputer system using previous basic modular structures.

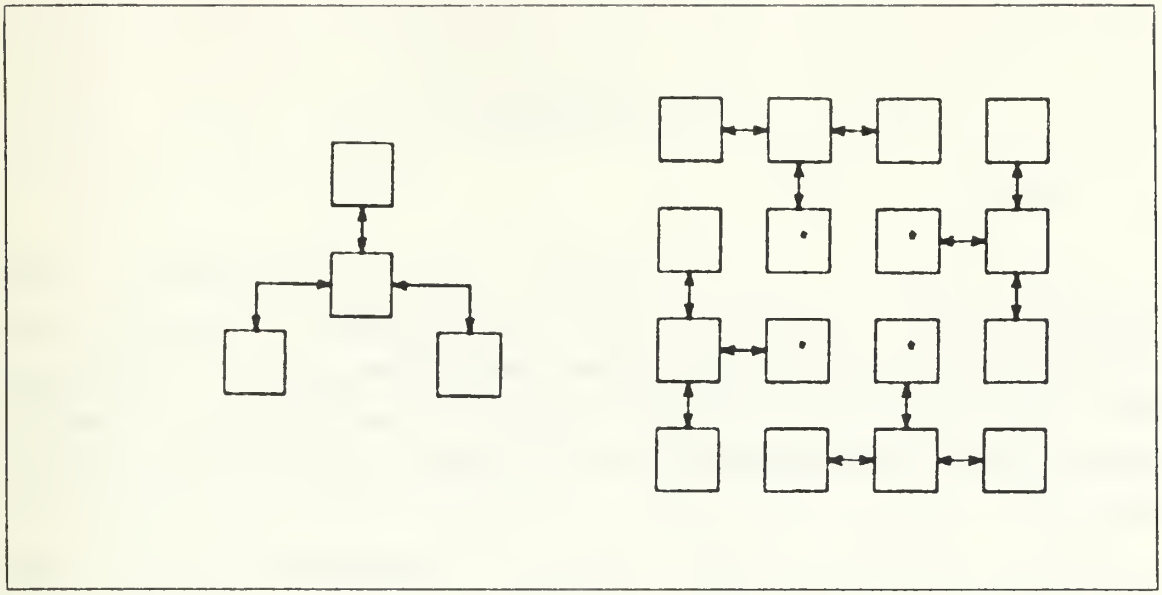

Figure 5.6 Star Type Computation Model

VI. CONCLUSION

A. SUMMARY

This thesis is focused on introducing the multitransputer system and its building blocks: the Transputer T424 and OCCAM programming language. Also the role of the fault tolerance in a real time system, is emphasised in the implementation of the multitransputer operating system.

The serial communication subsystem of the multitransputer operating system is implemented using the OCCAM programming language in the VAX 11/780 computer system.

Performance of the serial communication subsystem is evaluated and compared with a fault free system. As evaluated in Chapter V, the proposed multitransputer system performance is highly capable to achieve many present real time applications and a good candidate for future applications. This system becomes more attractive with its fault tolerance capability for many real time applications, especially in military based applications.

As far as the system dependability is concerned, the serial communication construct will provide a very high degree of probability of system functionality even if the parallel bus fails. Graceful degradation will be provided by sixteen computing elements in the multitransputer system. Also a common database application is possible during parallel bus failure.

The fault free operation is improved using triple or quadruple voting mechanisms which provides fault tolerance for many real time applications. This feature of the

multitransputer system can be also provided in the parallel bus failure case. In other words, the serial communication system has the capability of triple or quadruple voting mechanism applications.

B. FOLLOW-ON WORK

This thesis addressed only the serial communication subsystem for the fault tolerant multitransputer operating system. Possible continuation of this work include other subsystems of the proposed operating system, or the formalization of the serial communication subsystem using the ring communication structures.

APPENDIX A
COMMUNICATION PROCESSES

SUBPROCESS DECODER

```
PROC decode (CHAN receive.event, send.channel, xmit.channel)=
  WHILE TRUE
    receive.event ? ANY   -- activation signal from receiver
    ext.transputer := (ext.code\100)  -- determination of destination
    IF
      ext.transputer = transputer.number  -- correct destination ?
        SEQ                               -- if yes
          ext.index := (ext.code/100)     -- find index number
          send.channel ! ANY              -- activate sender process
          send.channel ? ANY              -- wait for synchronization
          receive.event ! ANY             -- sync signal to receiver

      ext.transputer <> transputer.number  -- incorrect destination ?
        SEQ                       -- if yes
          xmit.channel ! ANY      -- activate transmit process
          xmit.channel ? ANY      -- wait for synchronization
          receive.event ! ANY     -- sync signal to receiver
```

SUBPROCESS DUPLEXER

```
PROC duplexer (CHAN xmit.request, receive.off)=
  WHILE TRUE
     SEQ
       xmit.request ? ANY  -- transmition request
       receive.off  ! ANY  -- "receiver close" signal to receiver
       receive.off  ? ANY  -- "receiver closed" response from receiver
       xmit.request ! ANY  -- "transmit ready" response to transmitter
       xmit.request ? ANY  -- "transmit done" response from transmitter
       receive.off  ! ANY  -- "open receiver" signal to receiver
```


SUBPROCESS LISTEN

```
PROC listen (CHAN encoder.channel)=
  WHILE TRUE
    ALT i = [ 1 FOR 500 ]
      out.chan [i] ? int.data    -- message propagation from user process
        SEQ
          encoder.channel ! i    -- activation of encoder channel
          encoder.channel ? ANY -- sync signal from encoder channel
```

SUBPROCESS RECEIVE

```
PROC receive (CHAN decode.advance, receive.off) =
  VAR acknowledge, wait :
  WHILE TRUE
    SEQ
      wait := FALSE
      ALT
        link[1] ? ext.code     -- message from upper transputer
          active.link := 1
        link[2] ? ext.code     -- message from right transputer
          active.link := 2
        link[3] ? ext.code     -- message from lower transputer
          active.link := 3
        link[4] ? ext.code     -- message from left  transputer
          active.link := 4
        receive.off ? ANY      -- Duplexer "close receiver" request
          wait := TRUE
      IF
        NOT wait
          SEQ
            link[active.link] ! ext.code       -- handshake
            link[active.link] ? acknowledge    -- double handshake
            ext.code.type := FALSE
```

```
      IF
        acknowledge
          IF
            (ext.code/10000) = 0    -- determination of code type
              ext.code.type := TRUE
            (ext.code/10000) <> 0
              SEQ
                link[active.link] ? ext.data    -- receive data
                link[active.link] ! ext.data    -- handshake
                link[active.link] ? acknowledge -- double handshake
        TRUE
          SKIP
      IF
        acknowledge
          SEQ
            decode.advance ! ANY  -- activation of decoder channel
            decode.advance ? ANY  -- sync signal from decoder
        NOT acknowledge
          fault.links[receive.trouble] ! active.link
          -- activation of fault tolerance system
          -- in case of faulty communication
  wait
    SEQ
      receive.off ! ANY  -- "receiver closed" message to duplexer
      receive.off ? ANY  -- "open receiver" signal from duplexer
```

SUBPROCESS ENCODER

```
PROC encoder (CHAN send.channel, xmit.channel, message.event)=
  WHILE TRUE
    SEQ
      message.event ? int.index  -- activation signal from user processes
      int.transputer := process.table [int.index] -- destination number
      IF
        int.transputer = transputer.number -- short-cut communication ?
          SEQ              -- if yes
            send.channel  ! ANY  -- activate sender process
            send.channel  ? ANY  -- wait for synchronization signal
            message.event ! ANY  -- sync signal to listener process
        int.transputer <> transputer.number -- external communication ?
          SEQ              -- if yes
            IF
              int.index > 99             -- determine code type
                int.code.type := FALSE
              int.index <= 99
                int.code.type := TRUE
            int.code := (int.index * 100) + int.transputer  -- determine code
            xmit.channel  ! ANY  -- activate transmitter process
            xmit.channel  ? ANY  -- wait for synchronization signal
            message.event ! ANY  -- sync signal to listener process
```

SUBPROCESS SEND

```
PROC send (CHAN encoder.channel, decoder.channel)=
  WHILE TRUE
    ALT
      encoder.channel ? ANY  -- sending request from encoder (short-cut)
        IF
          int.code.type      -- if there is no data
            SEQ
              in.chan [int.index] ! ANY      -- send activation signal
              encoder.channel     ! ANY      -- sync signal to encoder
          NOT int.code.type  -- if there is a data
            SEQ
              in.chan [int.index] ! int.data -- send data
              encoder.channel     ! ANY      -- sync signal to encoder
      decoder.channel ? ANY  -- sending request from decoder
        IF
          ext.code.type      -- if there is no data
            SEQ
              in.chan [int.index] ! ANY      -- send activation signal
              decoder.channel     ! ANY      -- sync signal to decoder
          NOT ext.code.type   -- if there is a data
            SEQ
              in.chan [int.index] ! ext.data -- send data
              decoder.channel     ! ANY      -- sync signal to decoder
```

SUBPROCESS INTERFACE

```
PROC interface (CHAN xmit.advance, decode.chan, encoder.chan)=
  VAR xmit :
  WHILE TRUE
    ALT
      decode.chan ? ANY      -- eventcount from decoder channel
        SEQ
          XMIT := TRUE
          link.no := table1 [ext.transputer]    -- 1st priority path
          IF                                     -- available ?
            NOT (status[BYTE link.no] AND (link.no <> active.link))
              SEQ
                link.no := table2 [ext.transputer]  -- 2nd priority path
                IF                                   -- available ?
                  NOT (status[BYTE link.no] AND (link.no <> active.link))
                    SEQ
                      link.no := table3 [ext.transputer] -- 3rd priority path
                      IF                                 -- available ?
                        (status[BYTE link.no] AND (link.no <> active.link))
                          SKIP
                        TRUE
```

```
                    SEQ                    -- activation of fault tolerance
                      xmit := FALSE
                      fault.links[link.trouble] ! active.link
                      decode.chan ! go
            (status[BYTE link.no] AND (link.no <> active.link))
              SKIP
    (status[BYTE link.no] AND (link.no <> active.link))
      SKIP
  IF
    xmit      -- transmitting the message if there is no trouble
      SEQ
        code := ext.code
        data := ext.data
        code.type := ext.code.type
        xmit.advance ! link.no
        xmit.advance ? ANY    -- synchronization signal from transmitter
        decode.chan  ! ANY    -- synchronization signal to decoder
    NOT xmit  -- transmitting trouble
      SKIP
```

```
encoder.chan ? eventcount from encoder
  SEQ
    link.no := table1 [int.transputer]  -- 1st priority path
    IF
      NOT status[ BYTE link.no ]        -- available ?
        SEQ
          link.no := table2 [int.transputer] -- 2nd priority path
          IF
            NOT status [ BYTE link.no ]      -- available ?
              link.no := table3 [int.transputer] -- 3rd priority path
            TRUE
              SKIP
      TRUE
        SKIP
    code := int.code
    data := int.data
    code.type := int.code.type
    xmit.advance ! link.no    -- transmitting the message
    xmit.advance ? ANY    --  synchronization signal from xmitter
    encoder.chan ! ANY    --  synchronization signal to encoder
```

SUBPROCESS TRANSMITTER

```
PROC transmitter (CHAN xmit.event, xmit.request) =
  VAR temp.code, temp.data, temp.link, temp.code.type,
      response, fault :
  WHILE TRUE
    SEQ
      xmit.event ? temp.link  -- transmition eventcount
      temp.code := code
      temp.data := data
      temp.code.type := code.type
      xmit.event ! ANY        -- synchronization signal to interface
      xmit.request ! ANY      -- "receiver close" demand to duplexer
      xmit.request ? ANY      -- "receiver closed" message from duplexer
      link [temp.link] ! temp.code  -- transmitting the code
      link [temp.link] ? response   -- handshaking
      IF
        response = temp.code
          IF
            NOT temp.code.type
              SEQ
                link [temp.link] ! TRUE       -- double handshaking
                link [temp.link] ! temp.data  -- transmitting data
                link [temp.link] ? response   -- handshaking
```

```
            IF
              response = temp.data            -- control the handshake
                SEQ
                  link [temp.link] ! TRUE   -- double handshaking
                  fault := FALSE
              response <> temp.data           -- wrong response
                SEQ
                  link [temp.link] ! FALSE  -- double handshaking
                  fault := TRUE               -- transmit error is detected
        TRUE
          fault := FALSE   -- no error is detected
    response <> temp.code
      fault := TRUE        -- transmit error is detected
  IF
    fault      -- activation of fault tolerance system in case of error
      fault.links [xmit.trouble] ! ANY
    NOT fault
      SKIP
  xmit.request ! ANY   :   -- synchronization signal to duplexer
```

APPENDIX B

UTILITY PROCESSES

```
          ********************************************
          *                                          *
          *     GLOBAL VARIABLE DECLARATIONS         *
          *                                          *
          ********************************************

CHAN screen    AT 1 :
CHAN keyboard  AT 2 :
DEF  end.buffer := -3 ,
     bell := 7 ,
     cr :=  13 ,
     lf :=  10 :
VAR  location    [ BYTE   5 ] ,
     clear       [ BYTE   3 ] ,
     char.string [ BYTE 512 ] ,
     numero :

          ********************************************
          *                                          *
          *  VT100 TERMINAL INITIALIZATION VALUES    *
          *                                          *
          ********************************************

PROC init.screen =
  SEQ
    location  [BYTE 0] :=   4
    location  [BYTE 1] := 27
    location  [BYTE 2] := 'Y'
    clear     [BYTE 0] :=   2
    clear     [BYTE 1] := 27
    clear     [BYTE 2] := 'J'   :

          ********************************************
          *                                          *
          *    WRITING CHARACTER STRING TO CRT       *
          *                                          *
          ********************************************

PROC write.string ( VALUE string[] ) =
  SEQ
    SEQ i = [ 1 FOR string [ BYTE 0 ] ]
      screen ! string [ BYTE i ]
    screen ! end.buffer   :

          ********************************************
          *                                          *
          *     POSITIONING THE CURSOR ON CRT        *
          *                                          *
          ********************************************

PROC cursorXY ( VALUE x, y ) =
  SEQ location [ BYTE 3 ] := ' ' + y
  SEQ location [ BYTE 4 ] := ' ' + x
  write.string ( location ) :
```

```
            ********************************************
            *                                          *
            *      CLEAN SCREEN AND HOME CURSOR        *
            *                                          *
            ********************************************

PROC clear.screen =
  SEQ
    cursorXY (0,0)
    write.string (clear) :

            ********************************************
            *                                          *
            *      CARRIAGE RETURN / LINE FEED         *
            *                                          *
            ********************************************

PROC return =
  SEQ
    screen ! cr;lf;end.buffer  :

            ********************************************
            *                                          *
            *     WRITING DECIMAL NUMBER TO CRT        *
            *                                          *
            ********************************************

PROC write.number ( VALUE number ) =
  DEF minus := -1 :
  VAR k, j, negative, number,
      string [BYTE 15], temp[BYTE 15] :
  SEQ
    negative := FALSE
    IF
      number < 0
        SEQ
          number := minus * number
          negative := TRUE
      TRUE
        SKIP
    k := 0
    WHILE number <> 0
      SEQ
        temp [ BYTE k ] := ( number \ 10 ) + '0'
        k := k + 1
        number := number / 10
    SEQ i [ 2 FOR k+2 ]
      SEQ
        j := (k-i) + 1
        string [ BYTE i ] := temp [ BYTE j ]
    IF
      negative
        string [ BYTE 1 ] := '-'
      NOT negative
        string [ BYTE 1 ] := ' '
    string [ BYTE 0 ] := k+1
    write.string (string)  :
```

```
          ********************************************
          *                                          *
          *     READING DECIMAL NUMBER FROM K/B      *
          *                                          *
          ********************************************
PROC read.number =
  VAR input, sign, digits :
  SEQ
    keyboard ? input
    screen ! input;end.buffer
    sign := TRUE
    IF
      input = '-'
        input := '0'
        sign := FALSE
      input <> '-'
        SKIP
    digits := 0
    WHILE input <>  cr
      SEQ
        digits := ( digits * 10 ) + ( input - '0')
        keyboard ? input
        screen ! input;end.buffer
    IF
      NOT sign
        digits := -digits
      TRUE
        SKIP
    numero := digits  :

          ********************************************
          *                                          *
          *      READING STRING FROM KEYBOARD        *
          *                                          *
          ********************************************

PROC read.string =
  VAR n, char :
  SEQ
    n := 0
    keyboard ? char
    screen ! char;end.buffer
    WHILE char <> cr
      SEQ
        n := n + 1
        char.string [ BYTE n ] := char
        keyboard ? char
        screen ! char;end.buffer
    char.string [ BYTE 0 ] := n  :

          ********************************************
          *                                          *
          *           T I M E   D E L A Y            *
          *                                          *
          ********************************************

PROC delay ( VALUE microsec ) =
  VAR v :
  SEQ
    TIME ? v
    TIME ? AFTER (v+microsec*10) :
```

APPENDIX C
DISTANCE TABLES

The following three tables are used to determine the shortest available communication path between transputers. Tables have been designed to reflect matrix structure of the multi transputer system. Asterix '*' shows the propagation center. In order to find the shortest path at each priority level, we will need two input values. First propagation center which is determined by variable transputer.number for each transputer operating system. Second destination number which it can be found from last two digits of the code value. Actually each transputer operating system will include a single linear table, and destination number will be used as an index number of that table, in order to find the shortest communication path.

FIRST PRIORITY PATHS

*	2	2	4
3	3	3	4
1	2	3	4
1	2	4	1

4	*	2	2
4	3	3	3
4	1	2	3
1	1	2	4

2	4	*	2
3	4	3	3
3	4	1	2
4	1	1	2

2	2	4	*
3	3	4	3
2	3	4	1
2	4	1	1

1	2	4	1
*	2	2	4
3	3	3	4
1	2	3	4

1	1	2	4
4	*	2	2
4	3	3	3
4	1	2	3

4	1	1	2
2	4	*	2
3	4	3	3
3	4	1	2

2	4	1	1
2	2	4	*
3	3	4	3
2	3	4	1

1	2	3	4
1	2	4	1
*	2	2	4
3	3	3	4

4	1	2	3
1	1	2	4
4	*	2	2
4	3	3	3

3	4	1	2
4	1	1	2
2	4	*	2
3	4	3	3

2	3	4	1
2	4	1	1
2	2	4	*
3	3	4	3

3	3	3	4
1	2	3	4
1	2	4	1
*	2	2	4

4	3	3	3
4	1	2	3
1	1	2	4
4	*	2	2

3	4	3	3
3	4	1	2
4	1	1	2
2	4	*	2

3	3	4	3
2	3	4	1
2	4	1	1
2	2	4	*

SECOND PRIORITY PATHS

*	1	4	1
2	2	2	3
3	3	4	1
4	1	2	4

1	*	1	4
3	2	2	2
1	3	3	4
4	4	1	2

4	1	*	1
2	3	2	2
4	1	3	3
2	4	4	1

1	4	1	*
2	2	3	2
3	4	1	3
1	2	4	4

4	1	2	4
*	1	4	1
2	2	2	3
3	3	4	1

4	4	1	2
1	*	1	4
3	2	2	2
1	3	3	4

2	4	4	1
4	1	*	1
2	3	2	2
4	1	3	3

1	2	4	4
1	4	1	*
2	2	3	2
3	4	1	3

3	3	4	1
4	1	2	4
*	1	4	1
2	2	2	3

1	3	3	4
4	4	1	2
1	*	1	4
3	2	2	2

4	1	3	3
2	4	4	1
4	1	*	1
2	3	2	2

3	4	1	3
1	2	4	4
1	4	1	*
2	2	3	2

2	2	2	3
3	3	4	1
4	1	2	4
*	1	4	1

3	2	2	2
1	3	3	4
4	4	1	2
1	*	1	4

2	3	2	2
4	1	3	3
2	4	4	1
4	1	*	1

2	2	3	2
3	4	1	3
1	2	4	4
1	4	1	*

THIRD PRIORITY PATHS

*	3	3	3
4	1	4	1
2	1	2	3
2	4	1	2

3	*	3	3
1	4	1	4
3	2	1	2
2	2	4	1

3	3	*	3
4	1	4	1
2	3	2	1
1	2	2	4

3	3	3	*
1	4	1	4
1	2	3	2
4	1	2	2

2	4	1	2
*	3	3	3
4	1	4	1
2	1	2	3

2	2	4	1
3	*	3	3
1	4	1	4
3	2	1	2

1	2	2	4
3	3	*	3
4	1	4	1
2	3	2	1

4	1	2	2
3	3	3	*
1	4	1	4
1	2	3	2

2	1	2	3
2	4	1	2
*	3	3	3
4	1	4	1

3	2	1	2
2	2	4	1
3	*	3	3
1	4	1	4

2	3	2	1
1	2	2	4
3	3	*	3
4	1	4	1

1	2	3	2
4	1	2	2
3	3	3	*
1	4	1	4

4	1	4	1
2	1	2	3
2	4	1	2
*	3	3	3

1	4	1	4
3	2	1	2
2	2	4	1
3	*	3	3

4	1	4	1
2	3	2	1
1	2	2	4
3	3	*	3

1	4	1	4
1	2	3	2
4	1	2	2
3	3	3	*

LIST OF REFERENCES

1. Ihara, H., Fukuoka, K., Kubo, Y., Yokota, S., "Fault Tolerant Computer System With Three Symmetric Computers", Proceeding of the IEEE, October 1978.

2. Avizienis, A., Fault Tolerance : The Long Range Solution to the Maintainability of Electronic Systems, Government Microelectronic Application Conference, 1978.

3. Avizienis, A., "Fault Tolerance The Survival Attribute of Digital Systems", Proceedings of the IEEE, October 1978.

4. INMOS Limited, IMS T424 Transputer Advance Information , 1984

5. INMOS Limited, OCCAM Programming Manual, 1983

6. INMOS Technical Note 5, The Design of Concurrent Systems, 1984

7. Hopkins, A., Fault Tolerant System Design: Broad Brush and Fine Printing, Fault Tolerant Computing Symposium (FTCS), 1975.

8. Weitzman, C., Distributed Micro/MiniComputer Systems Prentice-Hall, New York, 1981

9. Drake A. W., Fundamentals of Applied Probability Theory, McGraw-Hill Book Company, New York, 1967

INITIAL DISTRIBUTION LIST

No.		Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22341	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943	1
4.	Prof. Uno R. Kodres, Code 52 Kr Department of Computer Science Naval Postgraduate School Monterey, California 93943	3
5.	Prof. M. L. Cotton, Code 62 Cc Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943	1
6.	Zafer Selcuk Haznedar Hurmali sk. Gunes apt. 3/4 Bahcelievler, Istanbul Turkey	3
7.	Daniel Green, Code 20E Naval Surface Weapons Center Dahlgren, Virginia 22449	1
8.	CAPT J. Donegan, USN PMS 400B5 Naval Sea System Command Washington, D.C. 20362	1
9.	RCA AEGIS Depository RCA Corporation Government Systems Division Mail Stop 127-327 Moorestown, New Jersey 08057	1
10.	Library (Code E33-05) Naval Surface Weapons Center Dahlgren, Virginia 22449	1
11.	Dr. M. J. Gralia Applied Physics Laboratory John Hopkins Road Laurel, Maryland 20707	1
12.	Dana Small Code 8242, NOSC San Diego, California 92152	1

13. Dz.K.K.ligi
Personel Daire Bsk.ligi
Ankara, Turkey — 1

14. Dz.Harp Ok.K.ligi
Fen Bilimleri Bl. Bsk.ligi
Heybeliada, Istanbul, Turkey — 2

15. Istanbul Teknik Universitesi
Kutuphanesi
Istanbul, Turkey — 1

16. Bogazici Universitesi
Kutuphanesi
Istanbul, Turkey — 1

17. Orta Dogu Teknik Universitesi
Kutuphanesi
Ankara, Turkey — 1

18. Ercument Dokanakoglu
Koykahve sk. no:11/3
Pasabahce, Istanbul, Turkey — 1

19. Bekir Evin
SMC 2876 NPGS
Monterey, California 93943 — 1

20. Gokalp Bayramoglu
Yildiz Bloklar A-1 blok
kat 11 daire 49 Yenimahalle
Ankara, Turkey — 1

Thesis
S41242
c.1

Selcuk
Implementation of a serial communication process for a fault tolerant, real time, multitransputer operating system.